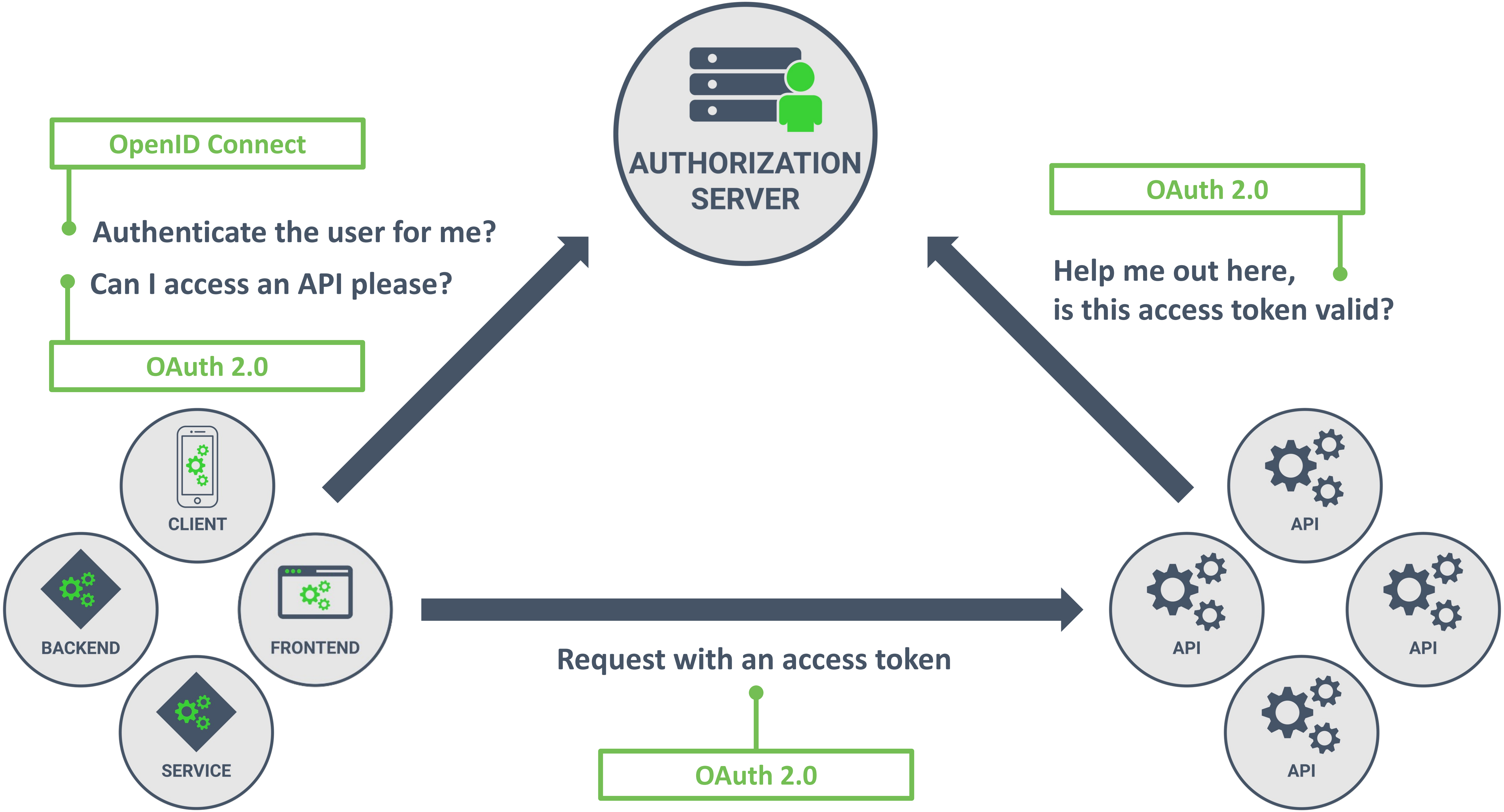
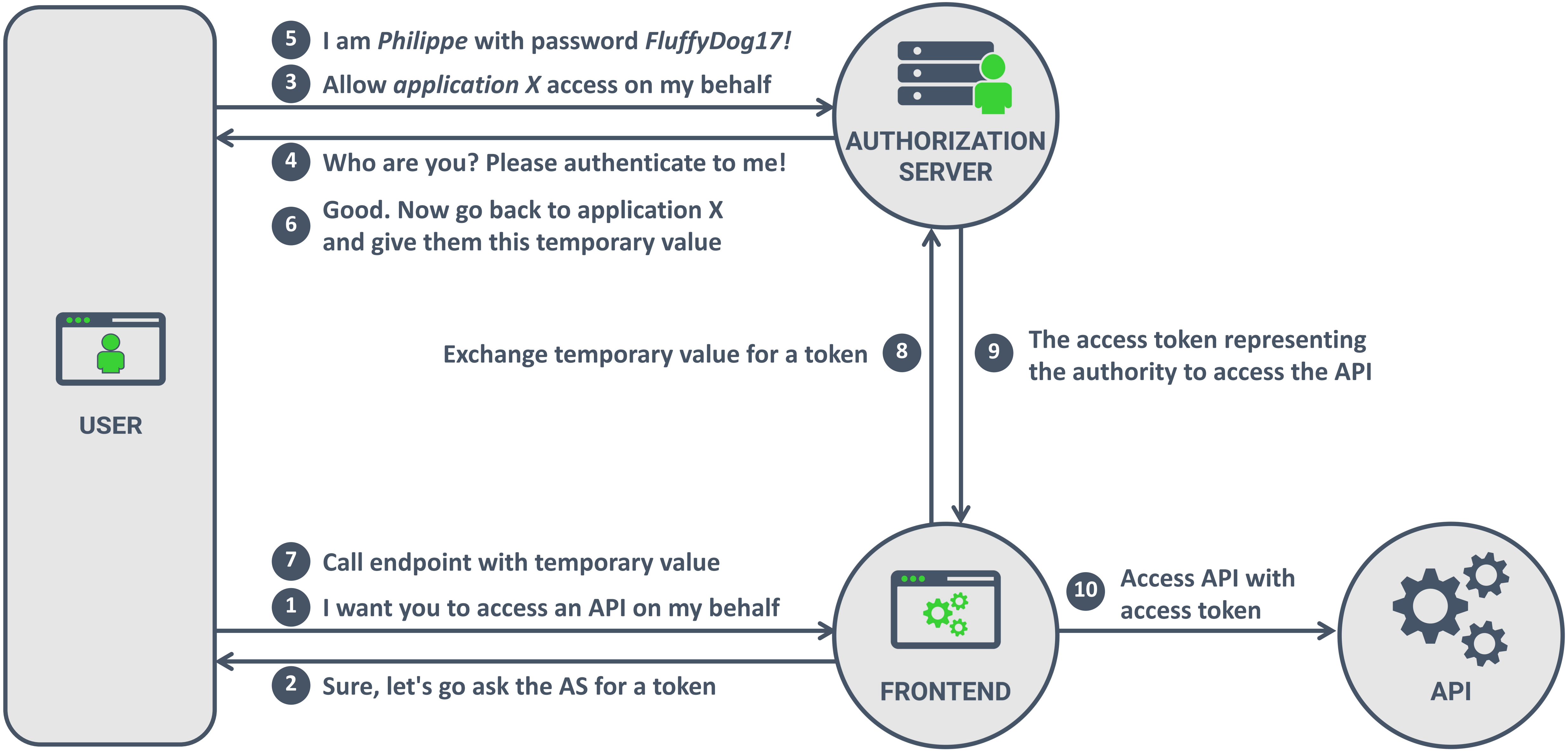
Breaking and securing OAuth 2.0 in frontends

Dr. Philippe De Ryck

https://PragmaticWebSecurity.com


AUTHORIZATION SERVER
OpenID Connect
Authenticate the user for me?
Can I access an API please?
OAuth 2.0
OAuth 2.0
Help me out here,
is this access token valid?
CLIENT
BACKEND
FRONTEND
SERVICE
Request with an access token
OAuth 2.0
API
API
API
API

THE CONCEPT OF THE OAUTH 2.0 AUTHORIZATION CODE FLOW



5 I am *Philippe* with password *FluffyDog17!*

3 Allow *application X* access on my behalf

4 Who are you? Please authenticate to me!

6 Good. Now go back to application X
and give them this temporary value

8 Exchange temporary value for a token

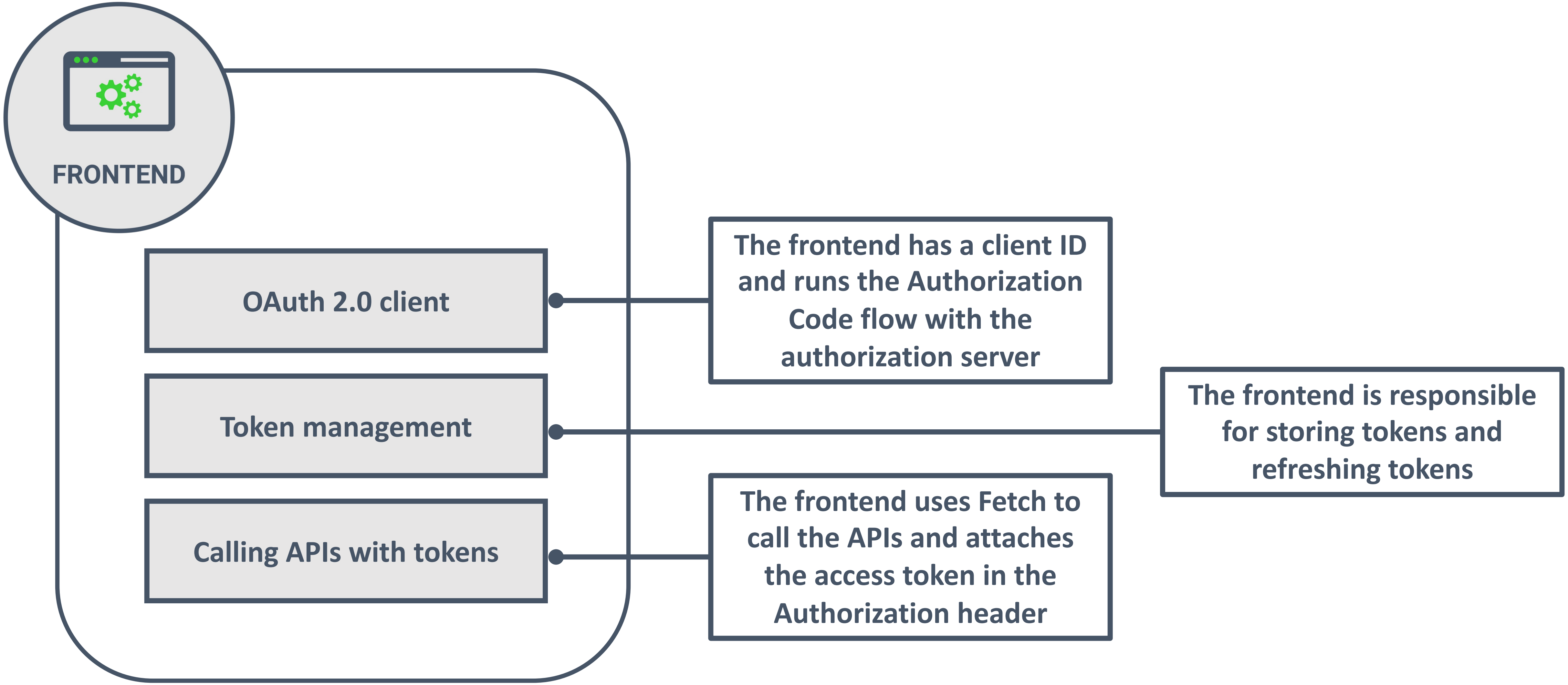
9 The access token representing
the authority to access the API

7 Call endpoint with temporary value

1 I want you to access an API on my behalf

2 Sure, let's go ask the AS for a token

10 Access API with
access token



FRONTEND

OAuth 2.0 client

Token management

Calling APIs with tokens

The frontend has a client ID and runs the Authorization Code flow with the authorization server

The frontend is responsible for storing tokens and refreshing tokens

The frontend uses Fetch to call the APIs and attaches the access token in the Authorization header

This pattern is a highly common practice for implementing OAuth 2.0 in frontends

I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers Experts

Google Developer Expert

SecAppDev

SecAppDev organizer

I help developers with security

- ✓ Hands-on in-depth security training
- ✓ Advanced online security courses
- ✓ Expert security advisory services

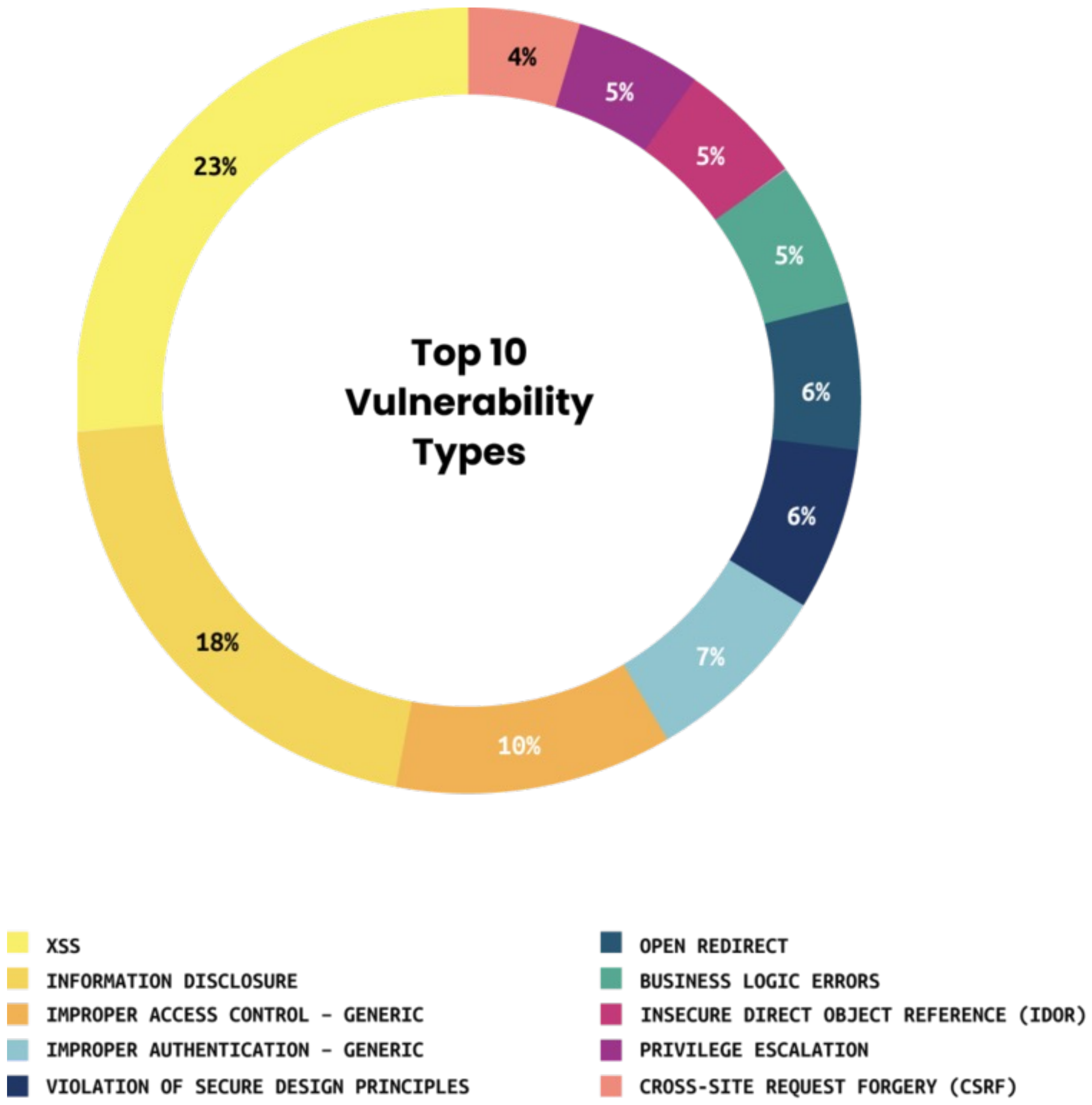
https://pdr.online

FRONTEND

JavaScript

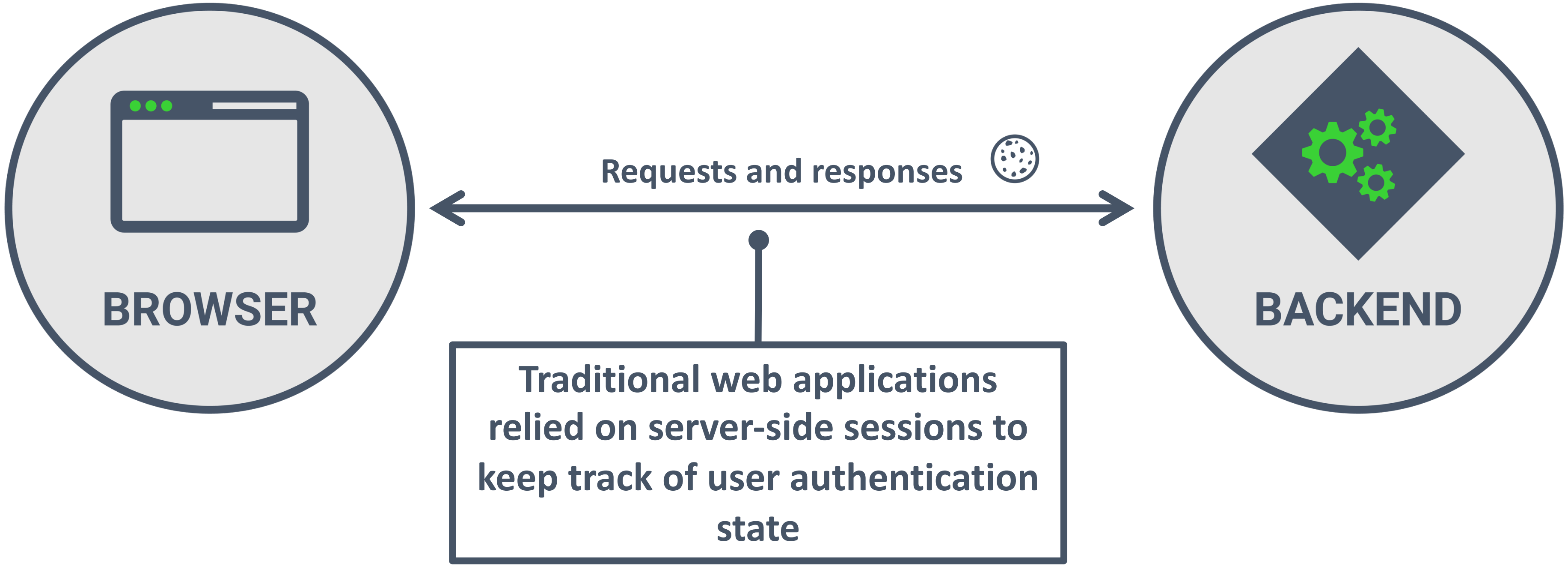
Malicious

JavaScript



https://www.hackerone.com/top-ten-vulnerabilities

XSS has been a problem for a long time

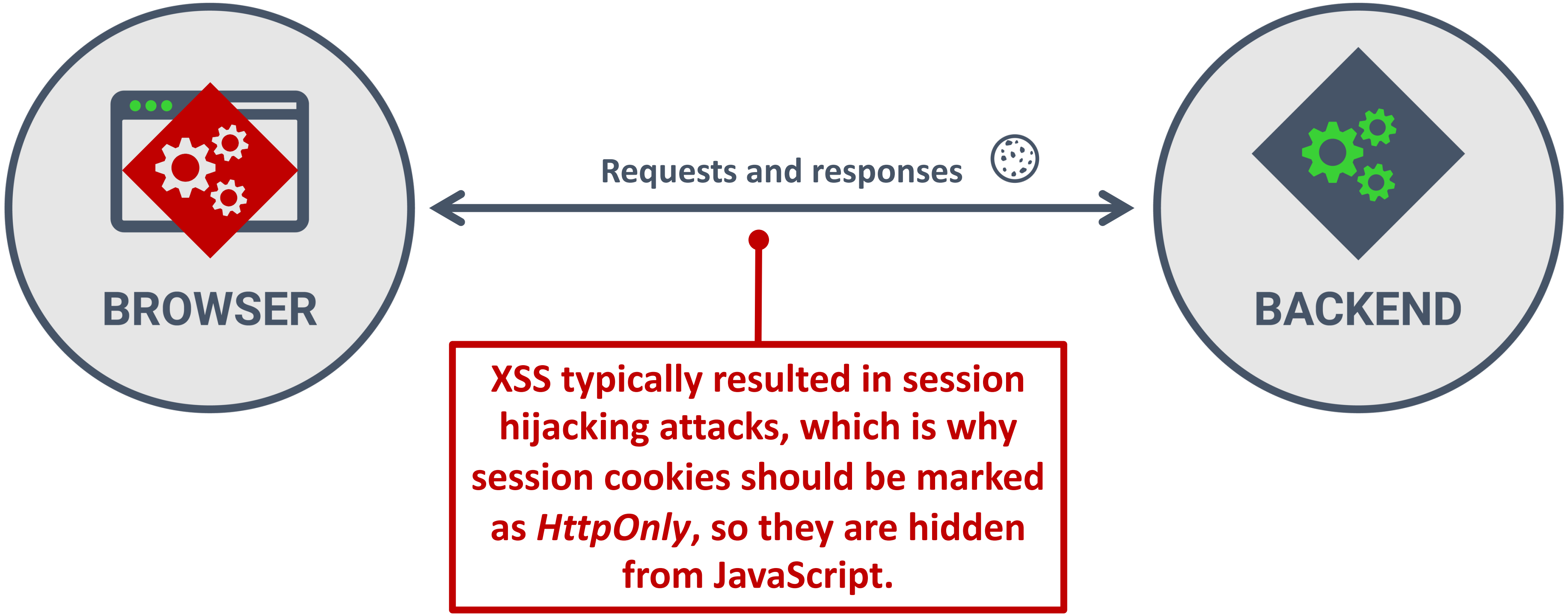


BROWSER

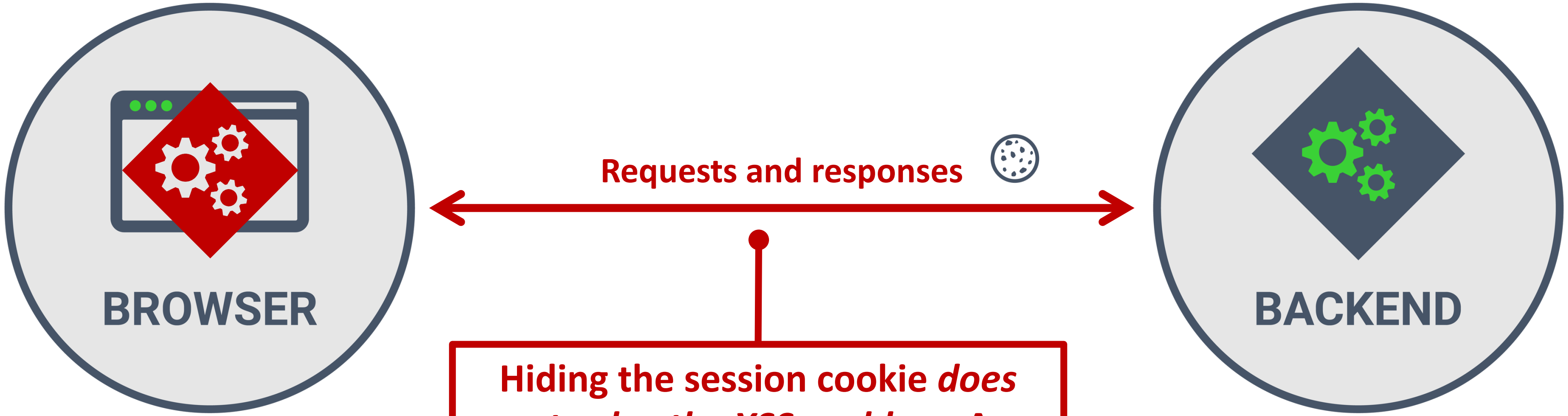
Requests and responses

BACKEND

Traditional web applications relied on server-side sessions to keep track of user authentication state



XSS typically resulted in session hijacking attacks, which is why session cookies should be marked as *HttpOnly*, so they are hidden from JavaScript.



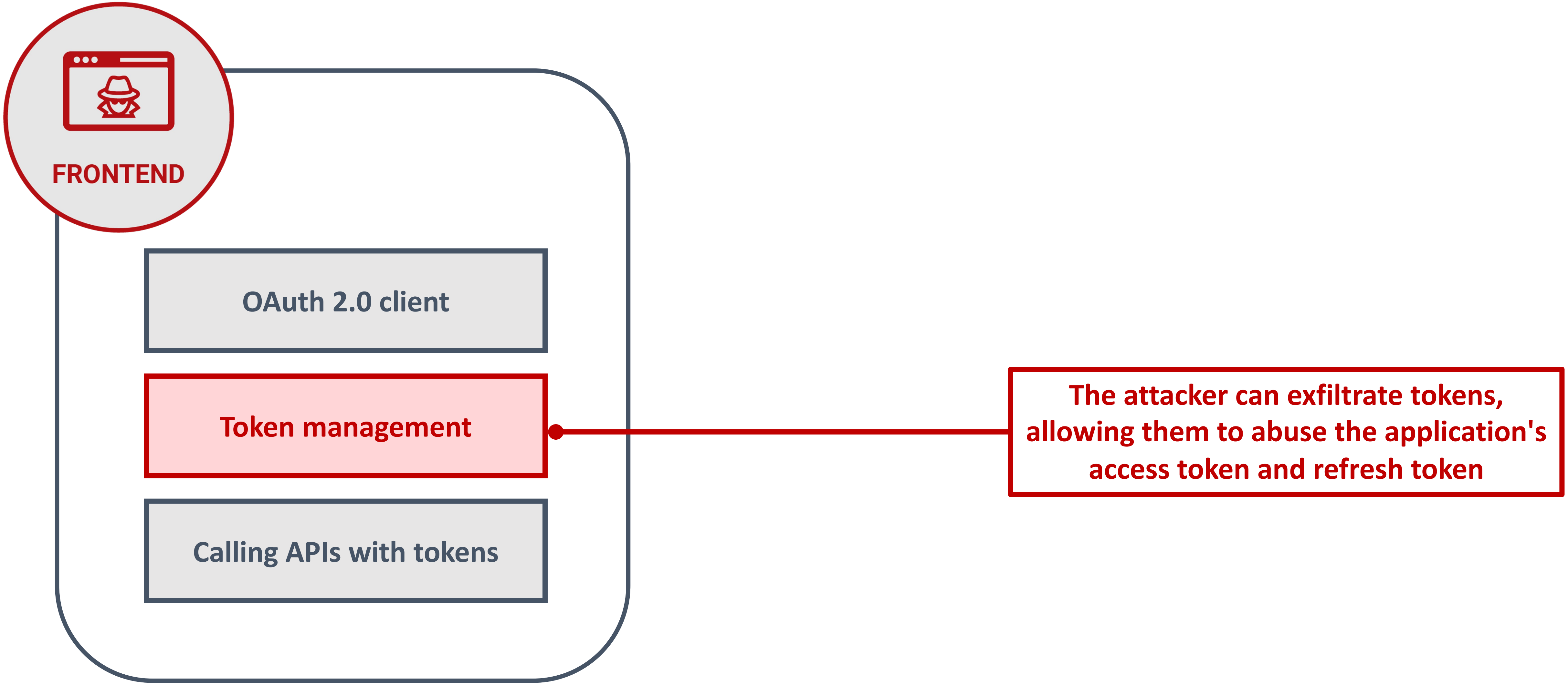
Hiding the session cookie *does not solve the XSS problem*. An attacker running code in the browser can still impersonate the user and manipulate the frontend.

XSS HAS ALWAYS BEEN A PROBLEM

Traditional web applications already suffered from XSS, with session hijacking as a common consequence.

Even then there was a misbelief that HttpOnly cookies *addressed the problem. However, once the malicious code runs, the attacker controls the client and can deceive or impersonate the user ...*

What does that mean for your OAuth 2.0 tokens?



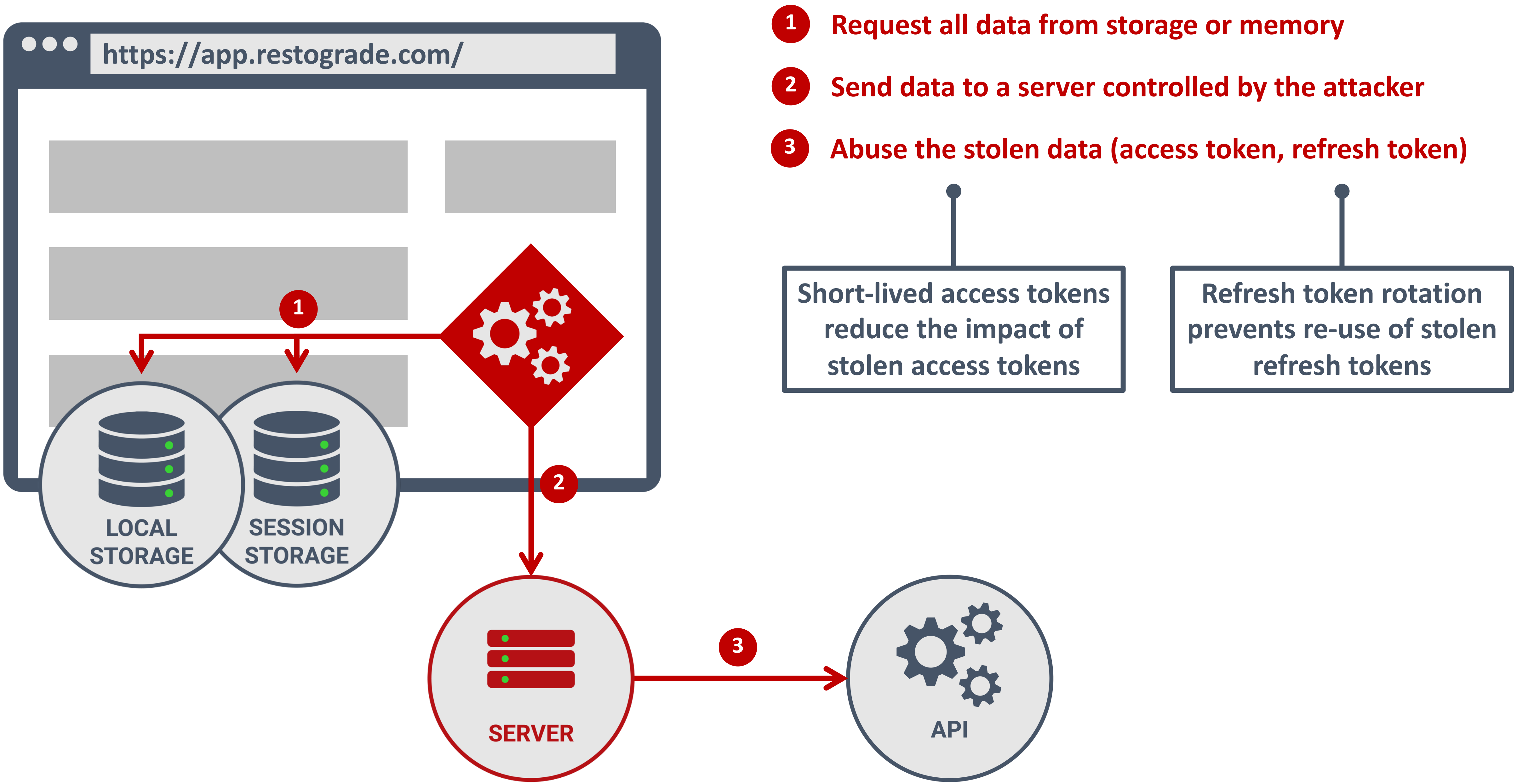
FRONTEND

OAuth 2.0 client

Token management

Calling APIs with tokens

The attacker can exfiltrate tokens, allowing them to abuse the application's access token and refresh token

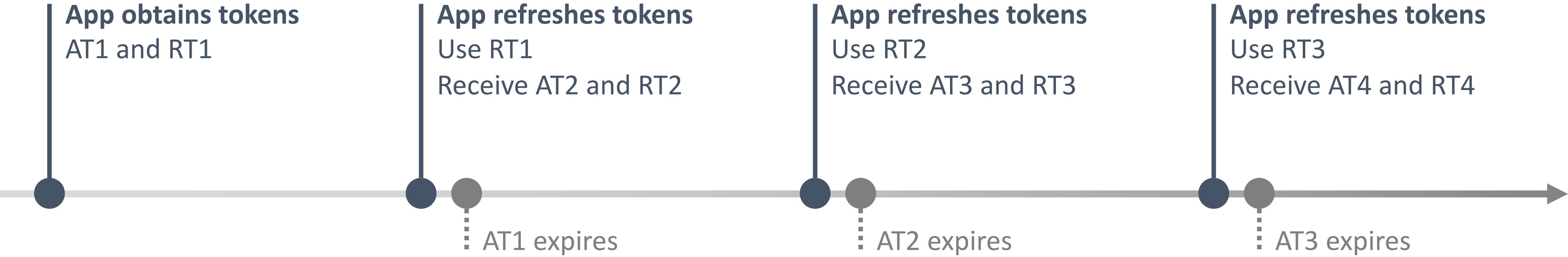


1. Request all data from storage or memory
2. Send data to a server controlled by the attacker
3. Abuse the stolen data (access token, refresh token)

Short-lived access tokens reduce the impact of stolen access tokens

Refresh token rotation prevents re-use of stolen refresh tokens

Refresh token rotation



App obtains tokens
AT1 and RT1

App refreshes tokens
Use RT1
Receive AT2 and RT2

AT1 expires

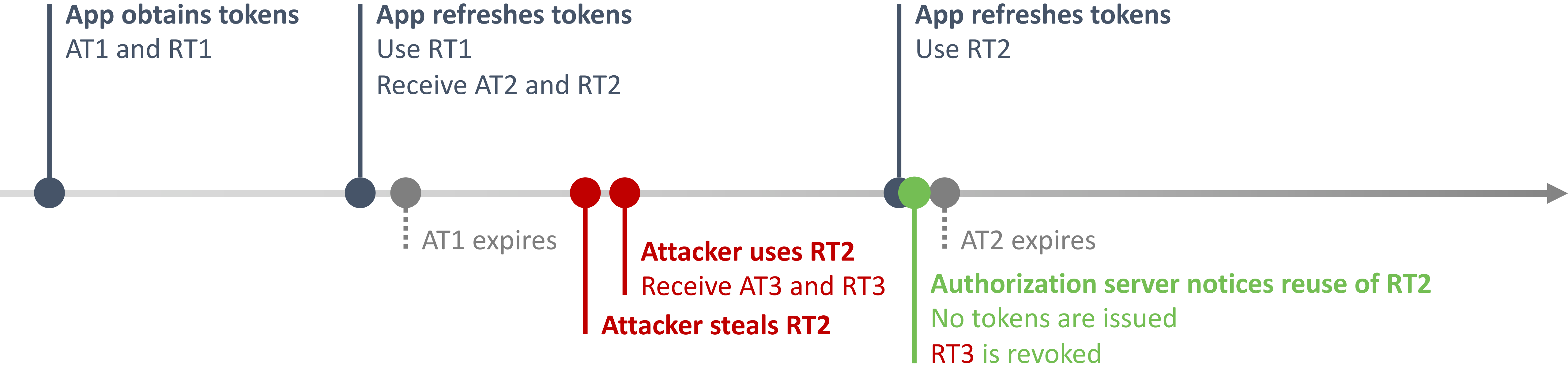
App refreshes tokens
Use RT2
Receive AT3 and RT3

AT2 expires

App refreshes tokens
Use RT3
Receive AT4 and RT4

AT3 expires

DETECTING REFRESH TOKEN ABUSE



App obtains tokens
AT1 and RT1

App refreshes tokens
Use RT1
Receive AT2 and RT2

AT1 expires

Attacker steals RT2

Attacker uses RT2
Receive AT3 and RT3

App refreshes tokens
Use RT2

AT2 expires

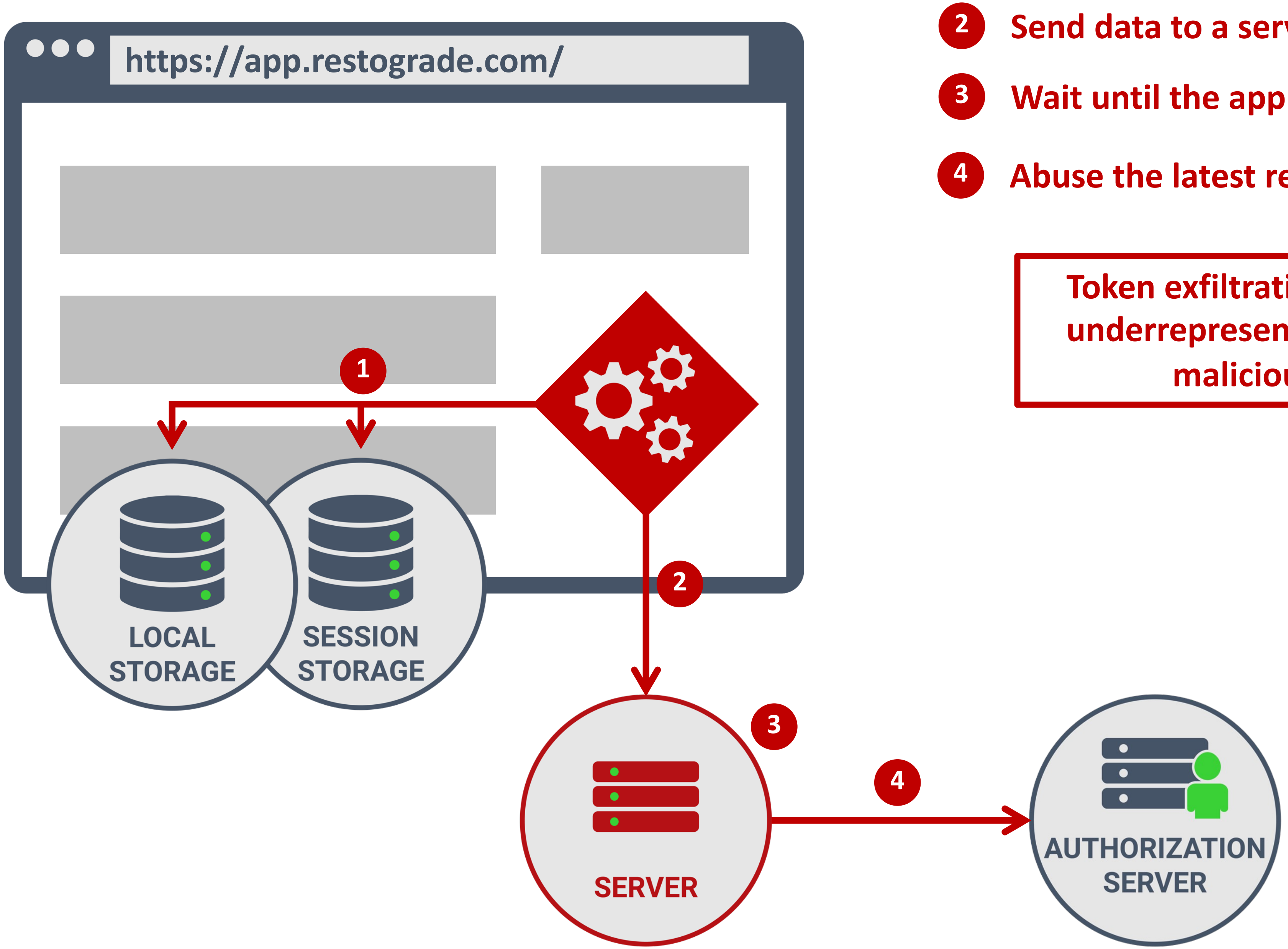
Authorization server notices reuse of RT2
No tokens are issued
RT3 is revoked

This is the common way of thinking, but this attacker representation severely underestimates the capabilities of the attacker

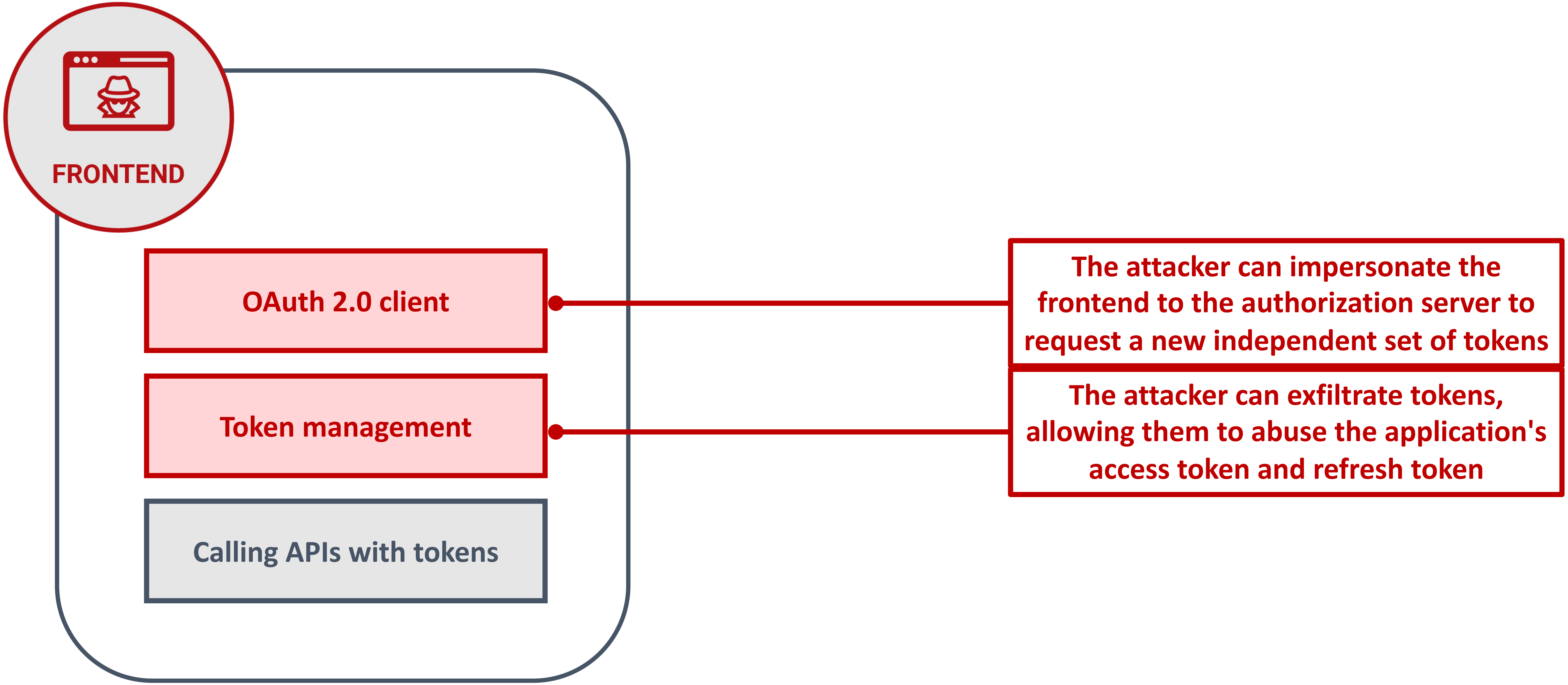
What happens with *Refresh Token Rotation* if a stolen refresh token is never used twice?

1. Request all data from storage or memory
2. Send data to a server controlled by the attacker
3. Wait until the application goes offline
4. Abuse the latest refresh token

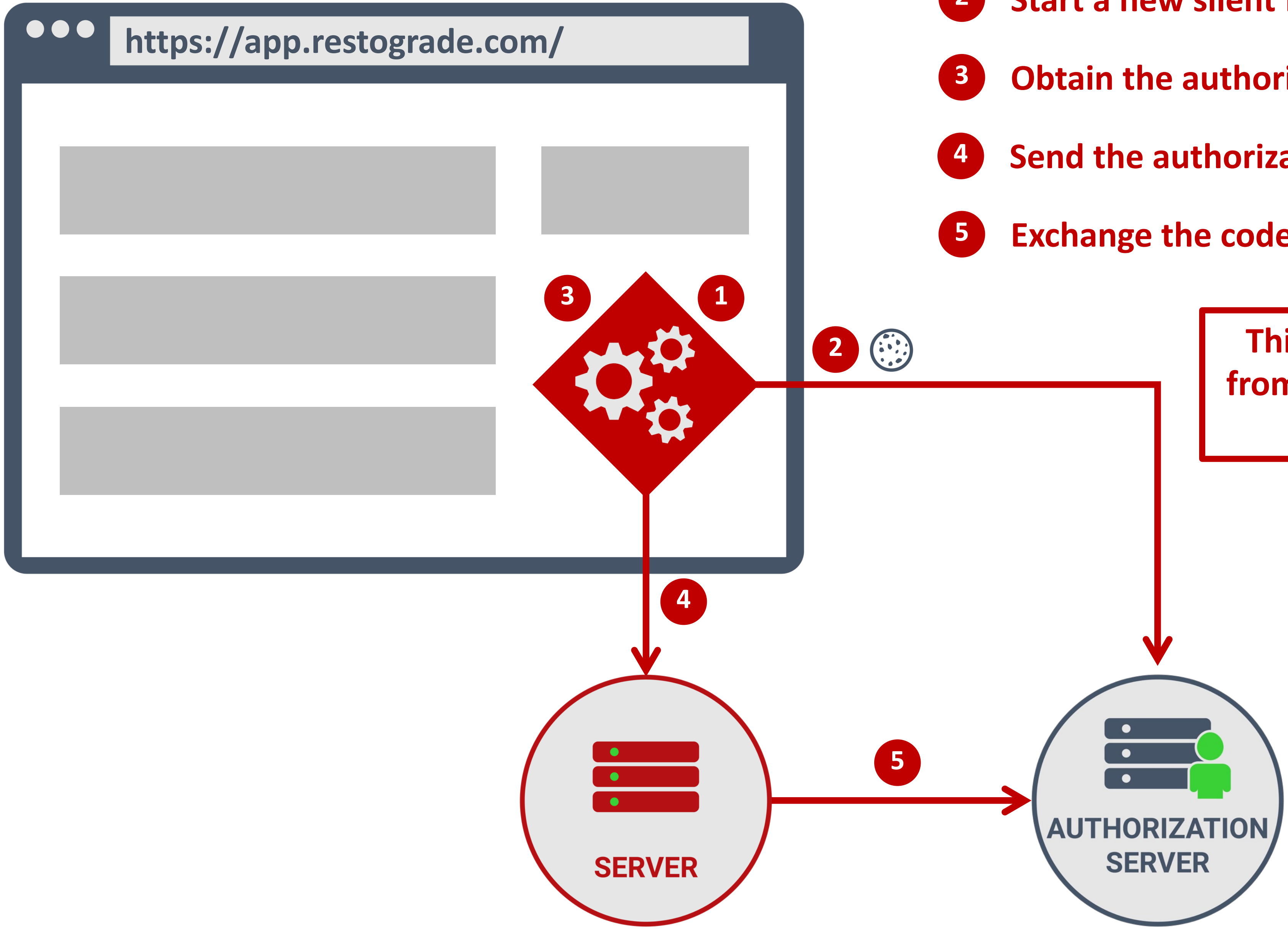
Token exfiltration attacks severely underrepresent the capabilities of malicious JavaScript



Sidestepping refresh token rotation


FRONTEND
OAuth 2.0 client
Token management
Calling APIs with tokens
The attacker can impersonate the frontend to the authorization server to request a new independent set of tokens
The attacker can exfiltrate tokens, allowing them to abuse the application's access token and refresh token

1. Setup a handler to receive a code from an iframe
2. Start a new silent flow in an iframe
3. Obtain the authorization code from the iframe
4. Send the authorization code to the attacker server
5. Exchange the code for a new set of tokens



This new set of tokens is independent from the application's tokens, so refresh token rotation does not help

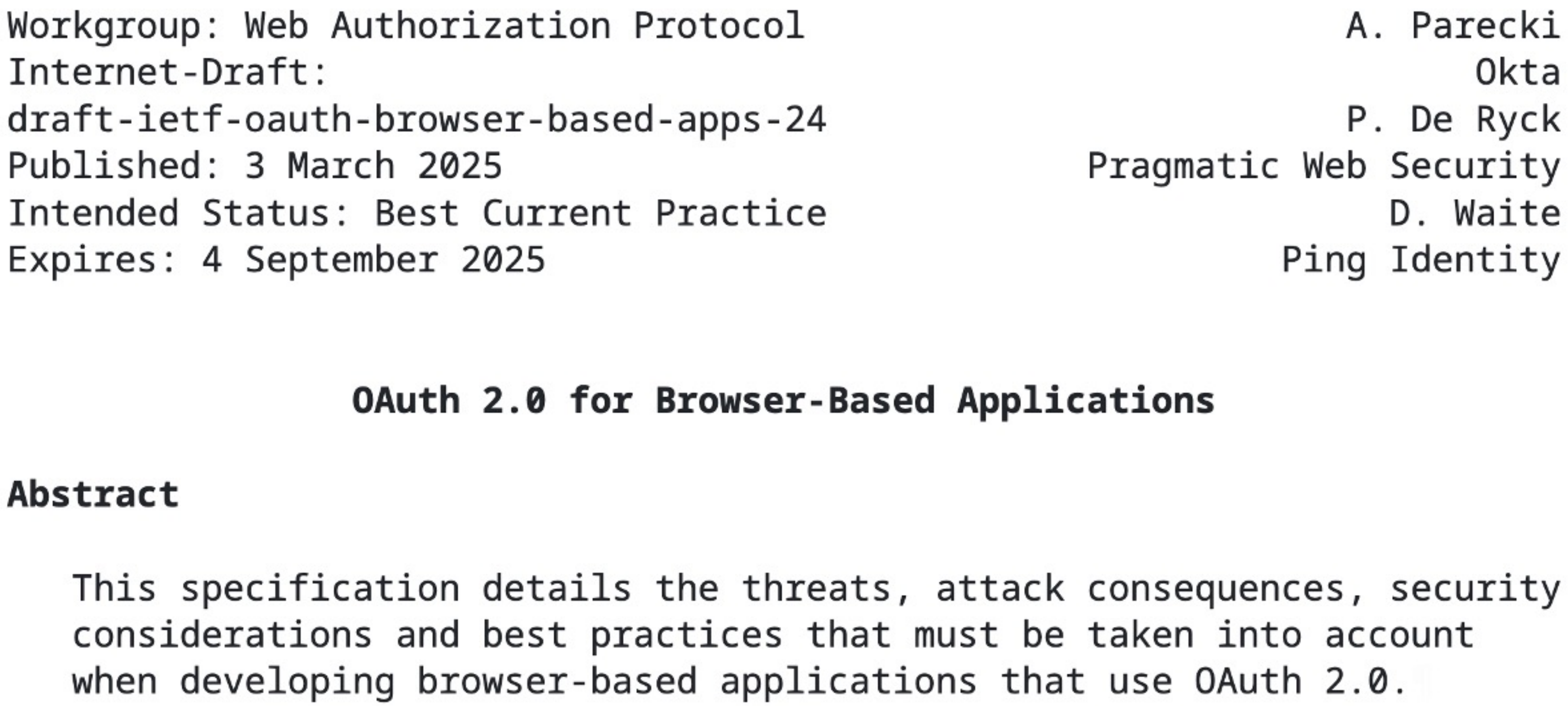
Requesting a fresh set of tokens

Additional security measures, such as DPoP do not work either, since the attacker can provide their own DPoP proofs

YOU CANNOT SECURE BROWSER-ONLY FLOWS

The security of OAuth 2.0 flows in the browser relies on the integrity of the frontend application and its origin (redirect URI).

When the attacker controls that origin, it's game over. Even proof-of-possession mechanisms cannot save you.



Workgroup: Web Authorization Protocol
Internet-Draft:
draft-ietf-oauth-browser-based-apps-24
Published: 3 March 2025
Intended Status: Best Current Practice
Expires: 4 September 2025

A. Parecki
Okta
P. De Ryck
Pragmatic Web Security
D. Waite
Ping Identity

OAuth 2.0 for Browser-Based Applications

Abstract

This specification details the threats, attack consequences, security considerations and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps.html

THREATS TO FRONTEND OAUTH 2.0 CLIENTS

Attack scenario	Example	Duration of attack
Single-execution token theft	One-time payload stealing an access token or refresh token from the running application	**Access tokens:** limited to token lifetime **Refresh tokens:** limited to detection with rotation
Persistent token theft	Continuously stealing access tokens or refresh tokens from the running application	**Access tokens:** as long as the user is online or the application is open **Refresh tokens:** limited to token lifetime after the user goes offline
Acquisition and extraction of new tokens	Running a silent Authorization Code flow to obtain a fresh access token and refresh token	The lifetime of the new refresh token (typically multiple hours or longer)
Proxying requests via the user's browser	Triggering API calls from within the frontend, authenticated by the application's access token	As long as the user is online or application is open

THREATS TO SERVER-SIDE OAUTH 2.0 CLIENTS

Attack scenario	Example	Duration of attack
~~**Single-execution token theft**~~	~~One-time payload stealing an access token or refresh token from the ru~~	~~**Access tokens:** limited to token lifetime~~ ~~**Refresh tokens:** limited to detection~~
~~**Persistent token theft**~~	~~Contin~~ ~~tokens or refresh tokens from the running application~~	~~is online~~ ~~**Refresh tokens:** limited to token lifetime after user goes offline~~
~~**Acquisition and extraction of new tokens**~~	~~Runni~~ ~~Code~~ ~~token~~	~~sh~~ ~~rs or~~
Proxying requests via the user's browser	Trigge fronte applic	

Server-side applications keep access tokens and refresh tokens in server-side storage (e.g., a database). An attacker executing malicious JS in the browser cannot access server-side token storage.

Server-side OAuth 2.0 clients need to authenticate their interactions with the authorization server, making it impossible for the attacker to exchange a stolen authorization code.

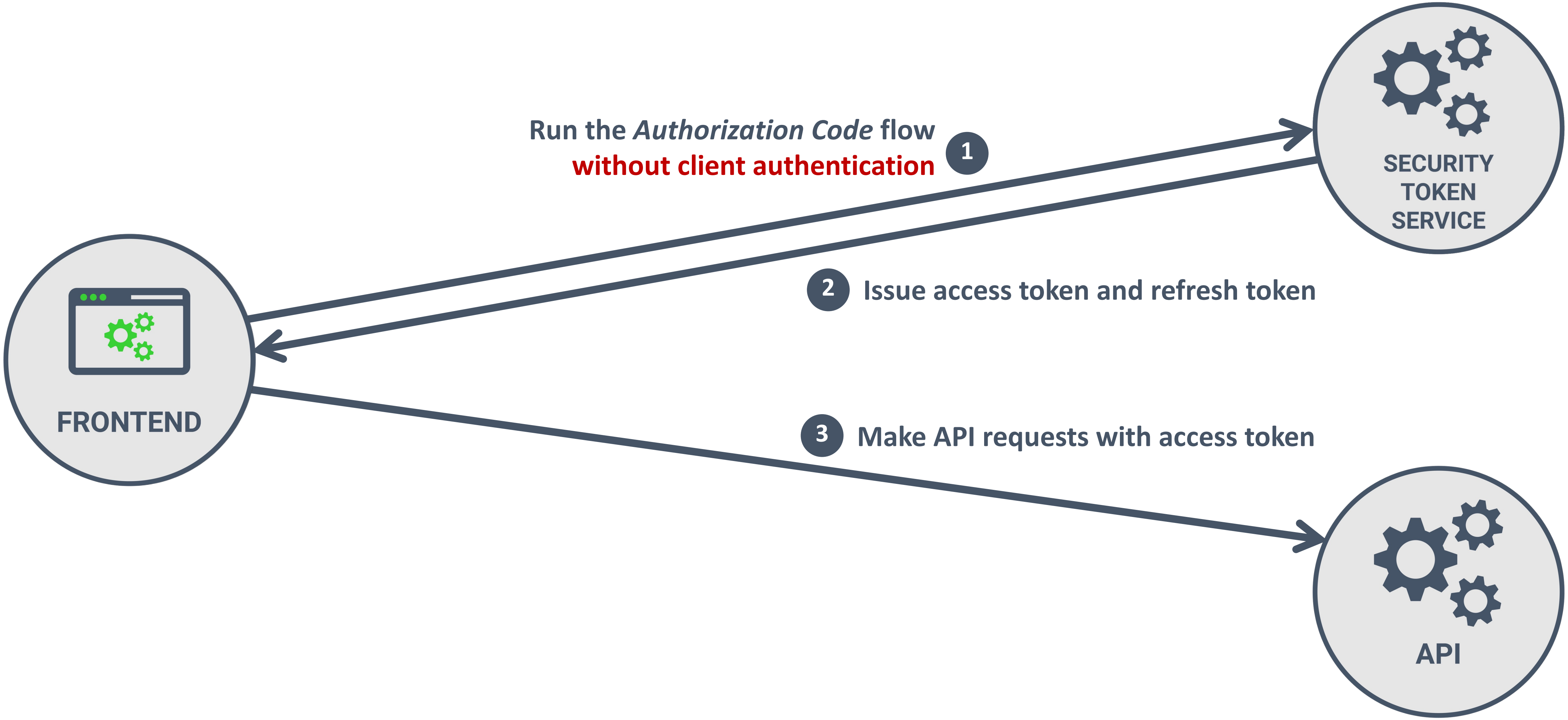
The attacker controlling pages in the browser can still send requests to the backend, which may result in data exfiltration or the execution of operations.

OAuth in frontends increases the attack surface

By using OAuth 2.0 in frontend applications, the attack surface of the application increases.

Attackers can impersonate the frontend application, allowing them to independently act in the name of the user for the lifetime of the refresh token.

Can we have the security of backend OAuth clients in our frontend applications?



Run the *Authorization Code* flow
without client authentication

1

2 Issue access token and refresh token

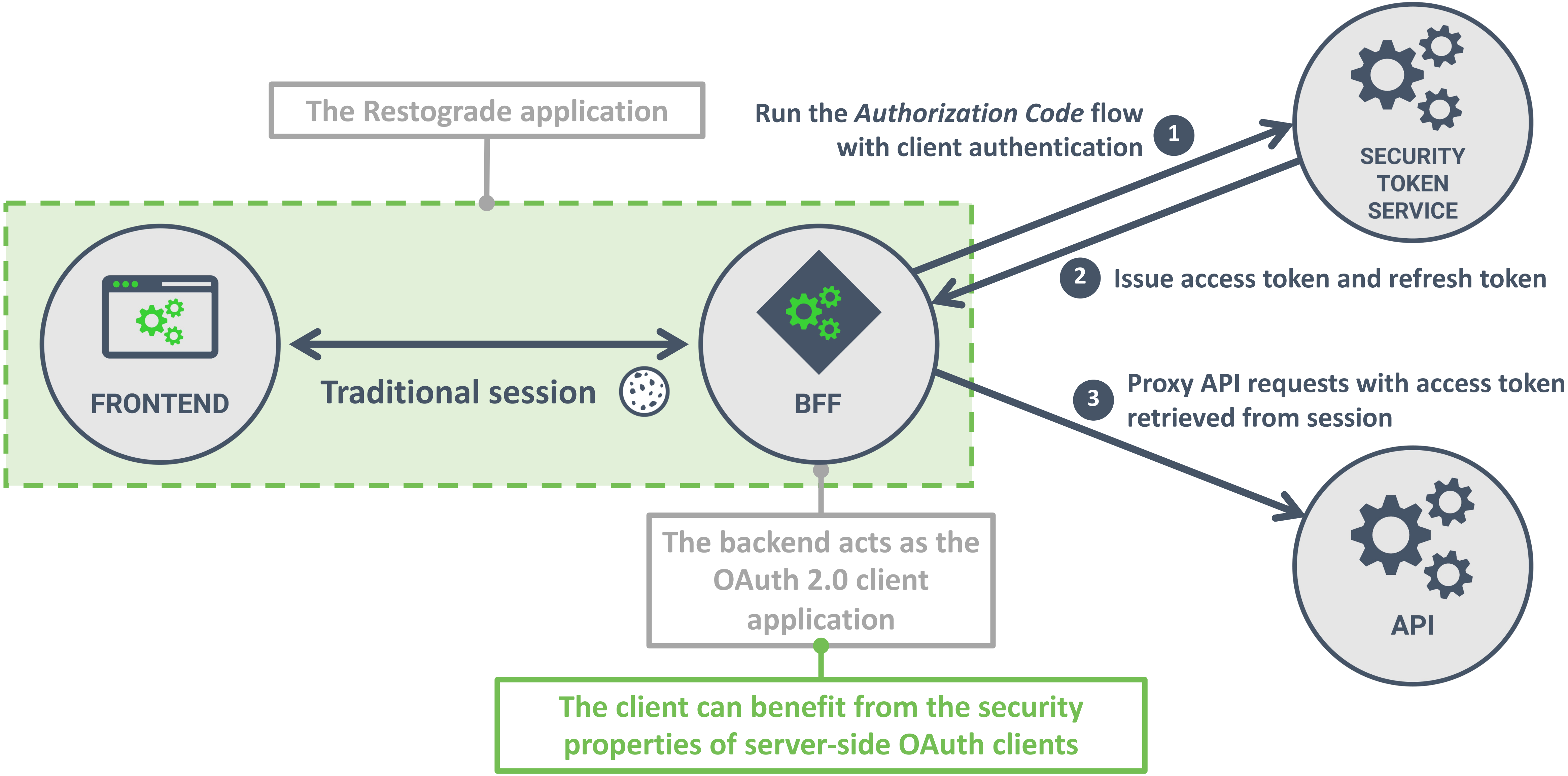
3 Make API requests with access token

SECURITY TOKEN SERVICE

FRONTEND

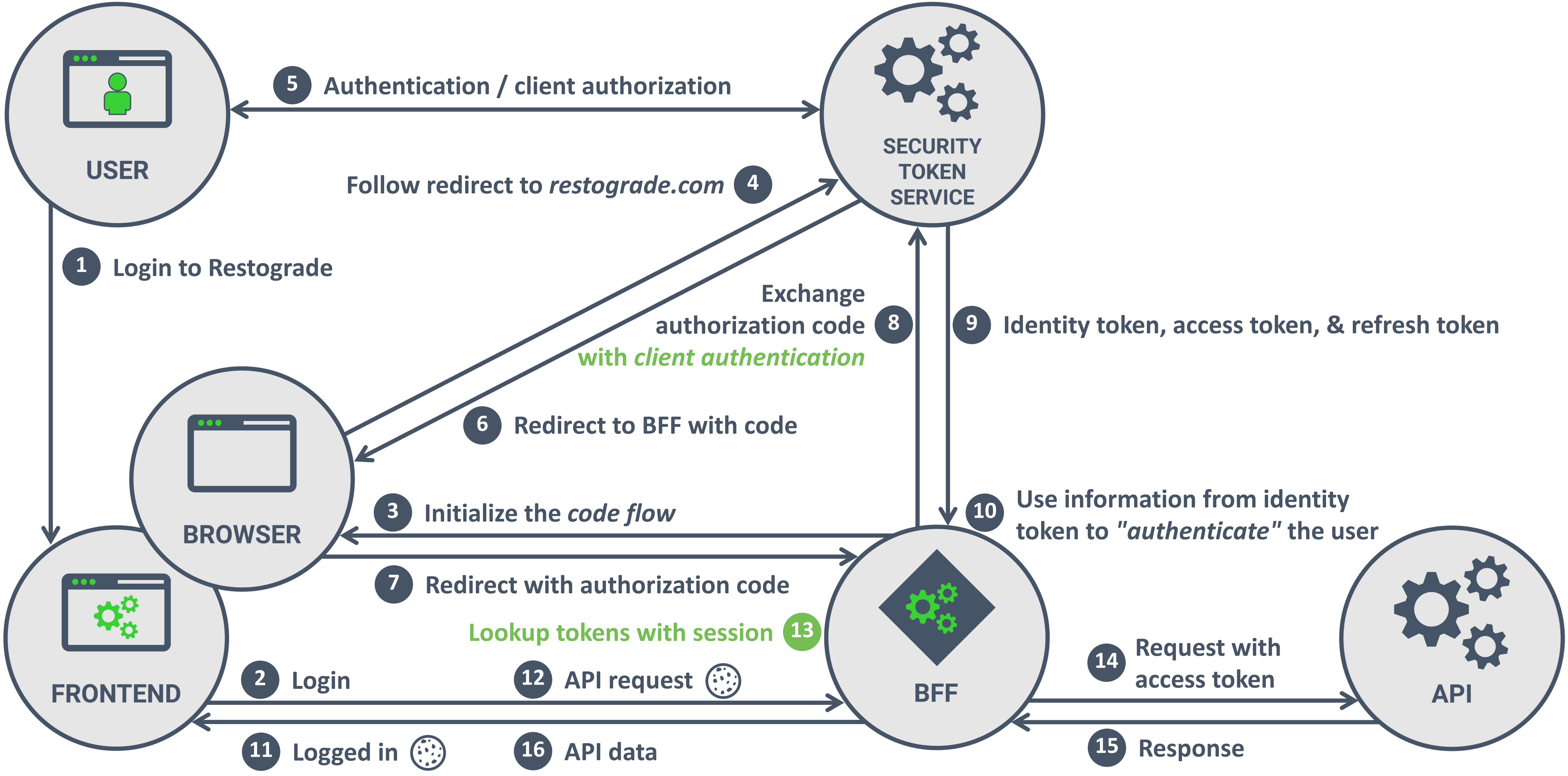
API

The concept of a backend-for-frontend



The Restograde application
Run the *Authorization Code* flow with client authentication
1
SECURITY TOKEN SERVICE
2
Issue access token and refresh token
FRONTEND
Traditional session
BFF
3
Proxy API requests with access token retrieved from session
API
The backend acts as the OAuth 2.0 client application
The client can benefit from the security properties of server-side OAuth clients

THE DETAILS OF A BACKEND-FOR-FRONTEND

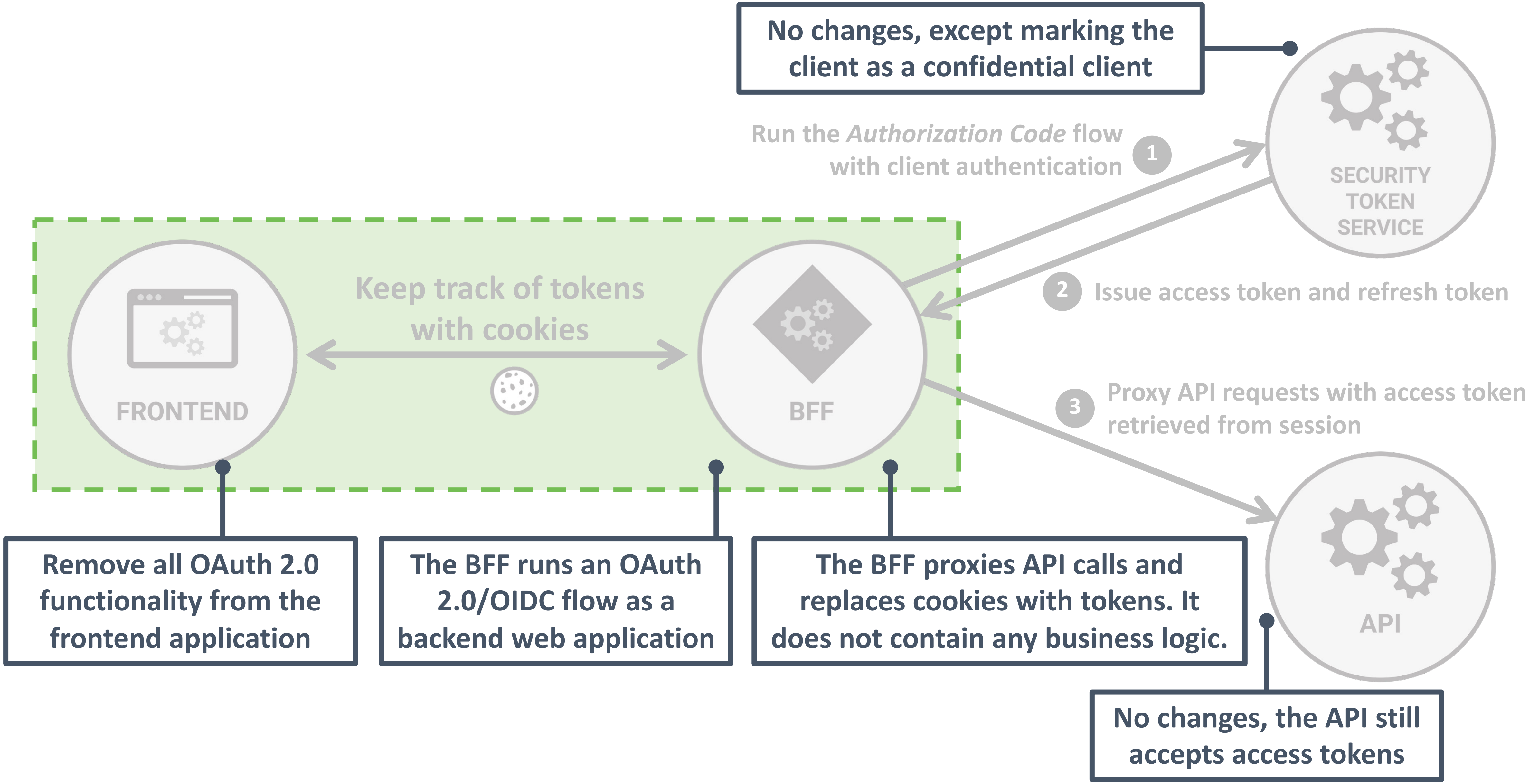

USER
SECURITY TOKEN SERVICE
BROWSER
FRONTEND
BFF
API
1 Login to Restograde
2 Login
3 Initialize the *code flow*
4 Follow redirect to *restograde.com*
5 Authentication / client authorization
6 Redirect to BFF with code
7 Redirect with authorization code
8 Exchange authorization code with *client authentication*
9 Identity token, access token, & refresh token
10 Use information from identity token to "*authenticate*" the user
11 Logged in
12 API request
13 Lookup tokens with session
14 Request with access token
15 Response
16 API data

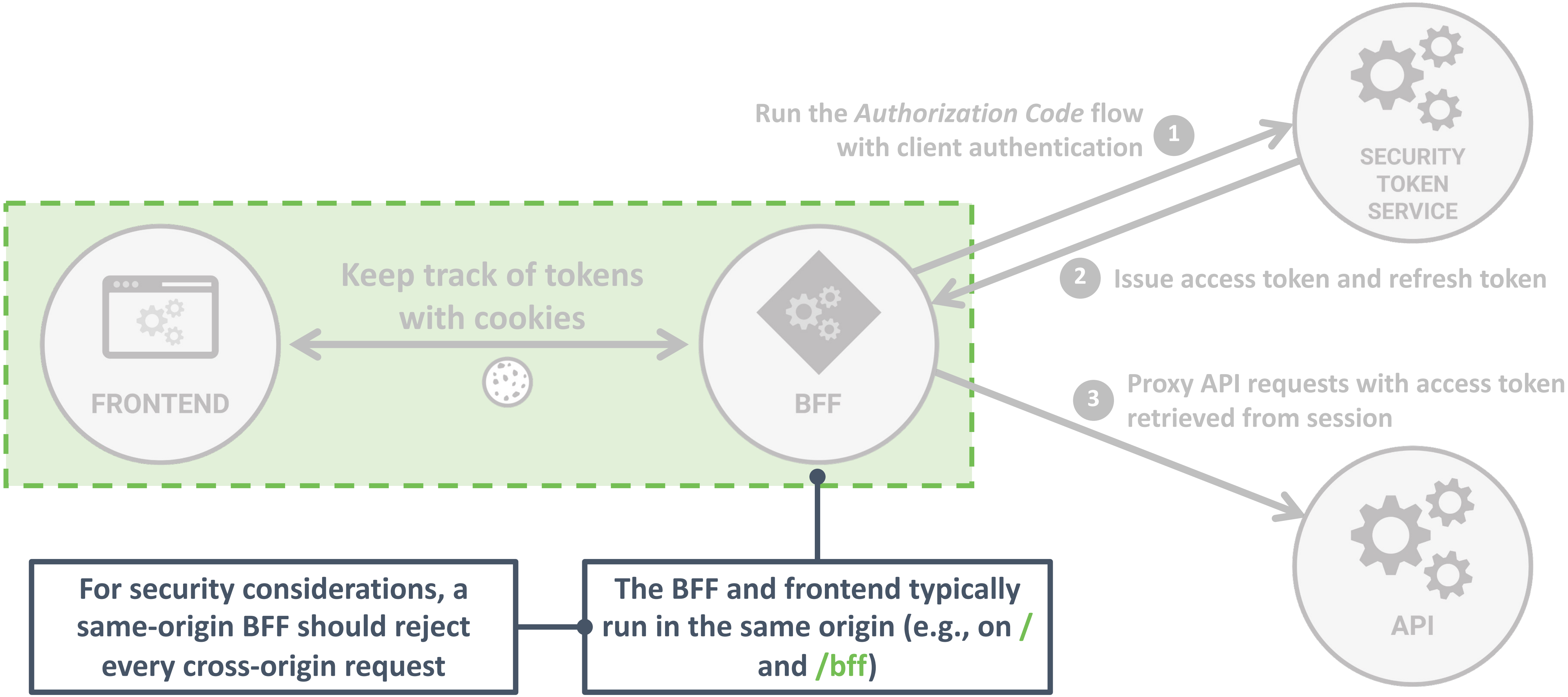
THE BACKEND-FOR-FRONTEND PATTERN

- The frontend uses a dedicated backend-for-frontend (BFF) for API access
  - The BFF mainly forwards calls to the actual APIs
  - The BFF attaches access tokens to outgoing requests to authorize the API calls

- BFFs are already used to aggregate different backend systems in a single API
  - Common pattern to join various microservices into a single frontend-specific API
  - Useful to chain different operations together without pushing that to the client
  - ***From a security perspective, BFFs make a lot of sense***

- The BFF becomes the OAuth 2.0 / OIDC client application
  - The BFF runs on a server, so it acts as a ***confidential client***
  - The BFF can apply all security best practices for backend client applications





Run the *Authorization Code* flow with client authentication

1

SECURITY TOKEN SERVICE

2 Issue access token and refresh token

Keep track of tokens with cookies

FRONTEND

BFF

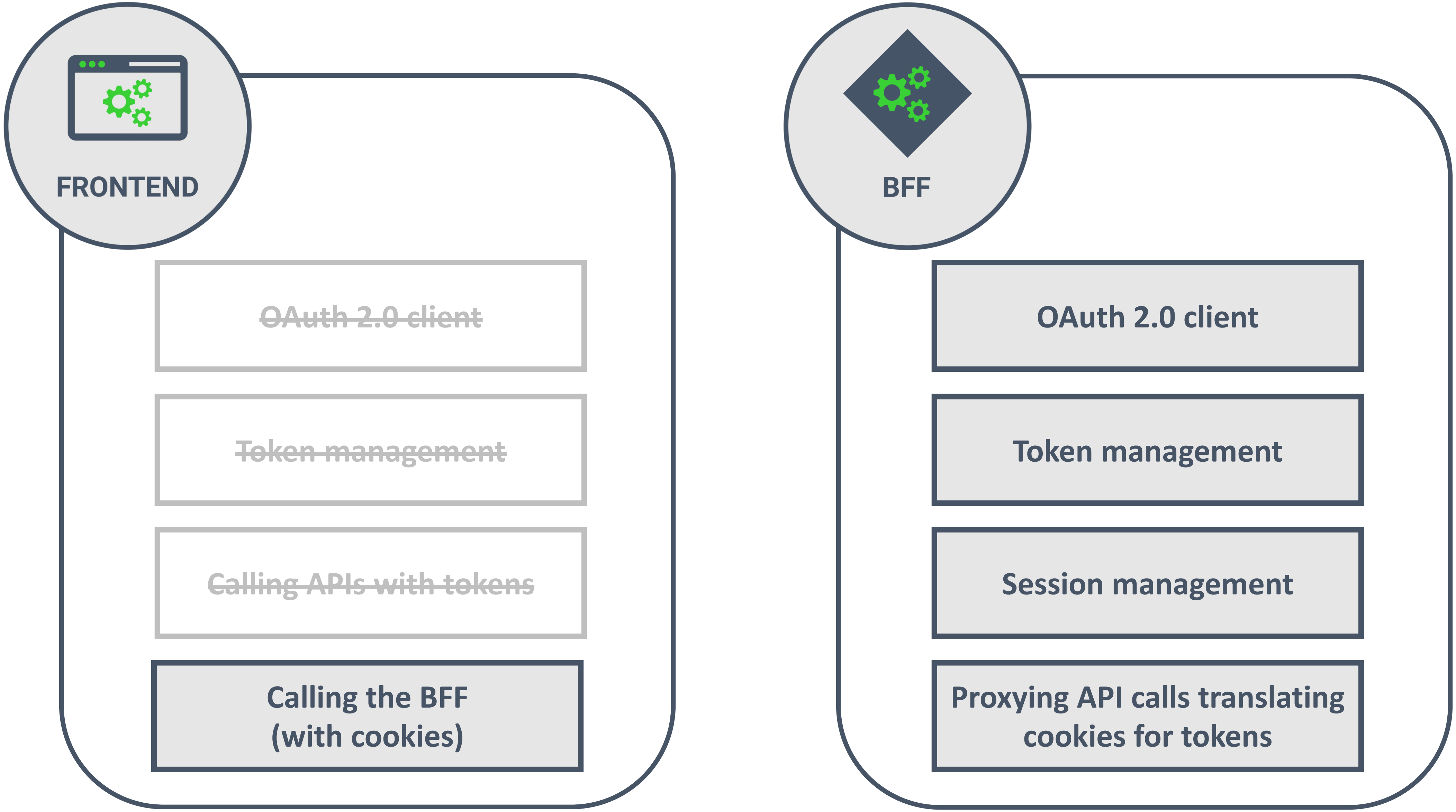
3 Proxy API requests with access token retrieved from session

API

For security considerations, a same-origin BFF should reject every cross-origin request

The BFF and frontend typically run in the same origin (e.g., on / and /bff)

A BFF in action



FRONTEND

~~OAuth 2.0 client~~

~~Token management~~

~~Calling APIs with tokens~~

Calling the BFF
(with cookies)

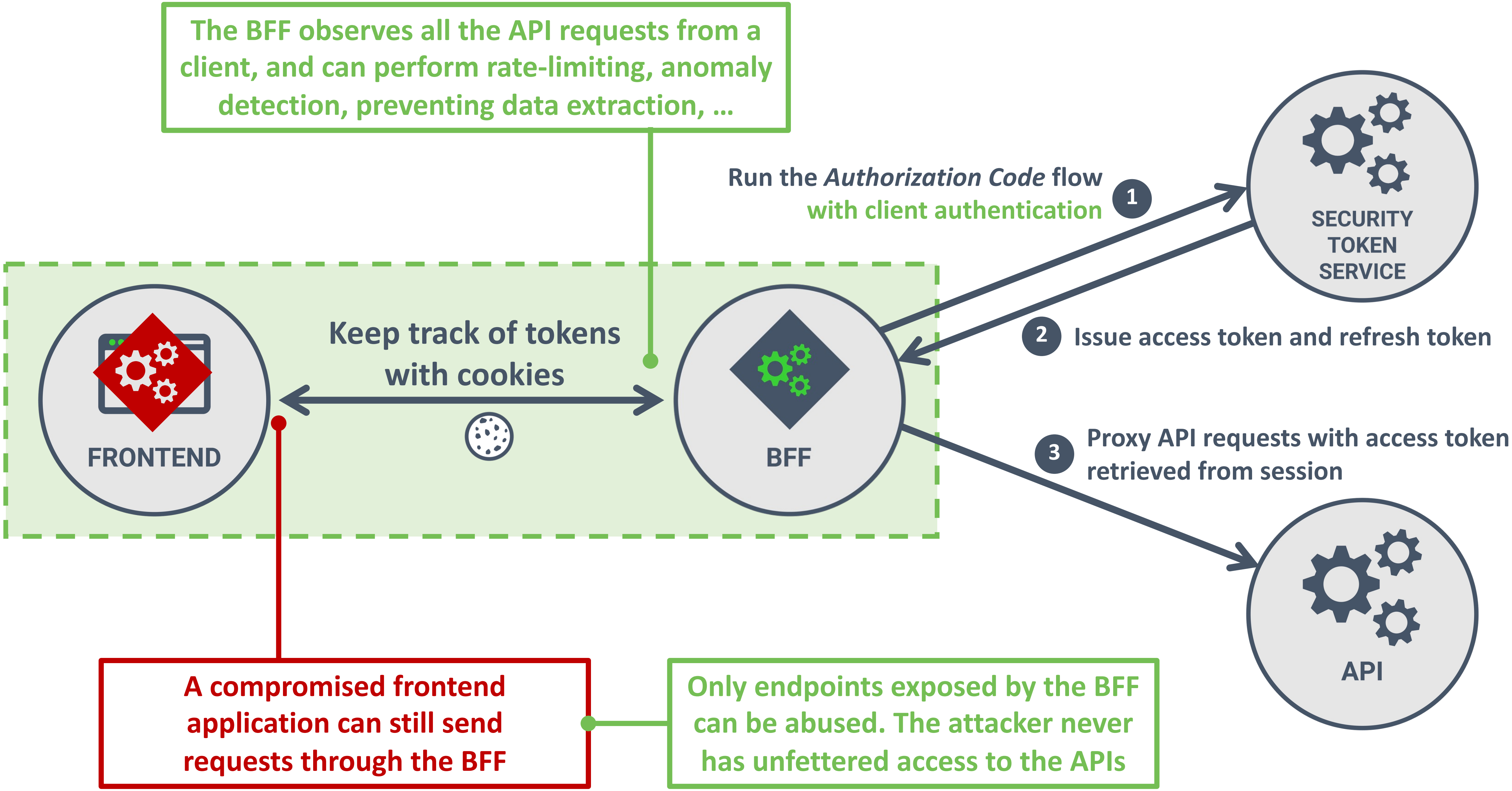
BFF

OAuth 2.0 client

Token management

Session management

Proxying API calls translating
cookies for tokens



The BFF observes all the API requests from a client, and can perform rate-limiting, anomaly detection, preventing data extraction, …

Keep track of tokens with cookies

1 Run the *Authorization Code* flow with client authentication

2 Issue access token and refresh token

3 Proxy API requests with access token retrieved from session

A compromised frontend application can still send requests through the BFF

Only endpoints exposed by the BFF can be abused. The attacker never has unfettered access to the APIs

THREATS TO FRONTENDS WITH A BFF

Attack scenario	Example	Duration of attack
~~**Single-execution token theft**~~	~~One-time payload stealing an access token or refresh token from the ru~~	~~**Access tokens:** limited to token lifetime~~ ~~**Refresh tokens:** limited to detection~~
~~**Persistent token theft**~~	~~Conti~~ ~~tokens or refresh tokens from the running application~~	~~is online or application is open~~ ~~**Refresh tokens:** limited to token lifetime after user goes offline~~
~~**Acquisition and extraction of new tokens**~~	~~Runni~~ ~~Code~~ ~~token~~	~~sh~~ ~~rs or~~
Proxying requests via the user's browser	Trigge fronte applic	

Only the BFF has access to the application's tokens, preventing an attacker executing malicious JS in the browser cannot access server-side token storage.

The BFF is a server-side OAuth 2.0 client using authentication on its interactions with the authorization server, making it impossible for the attacker to impersonate the BFF.

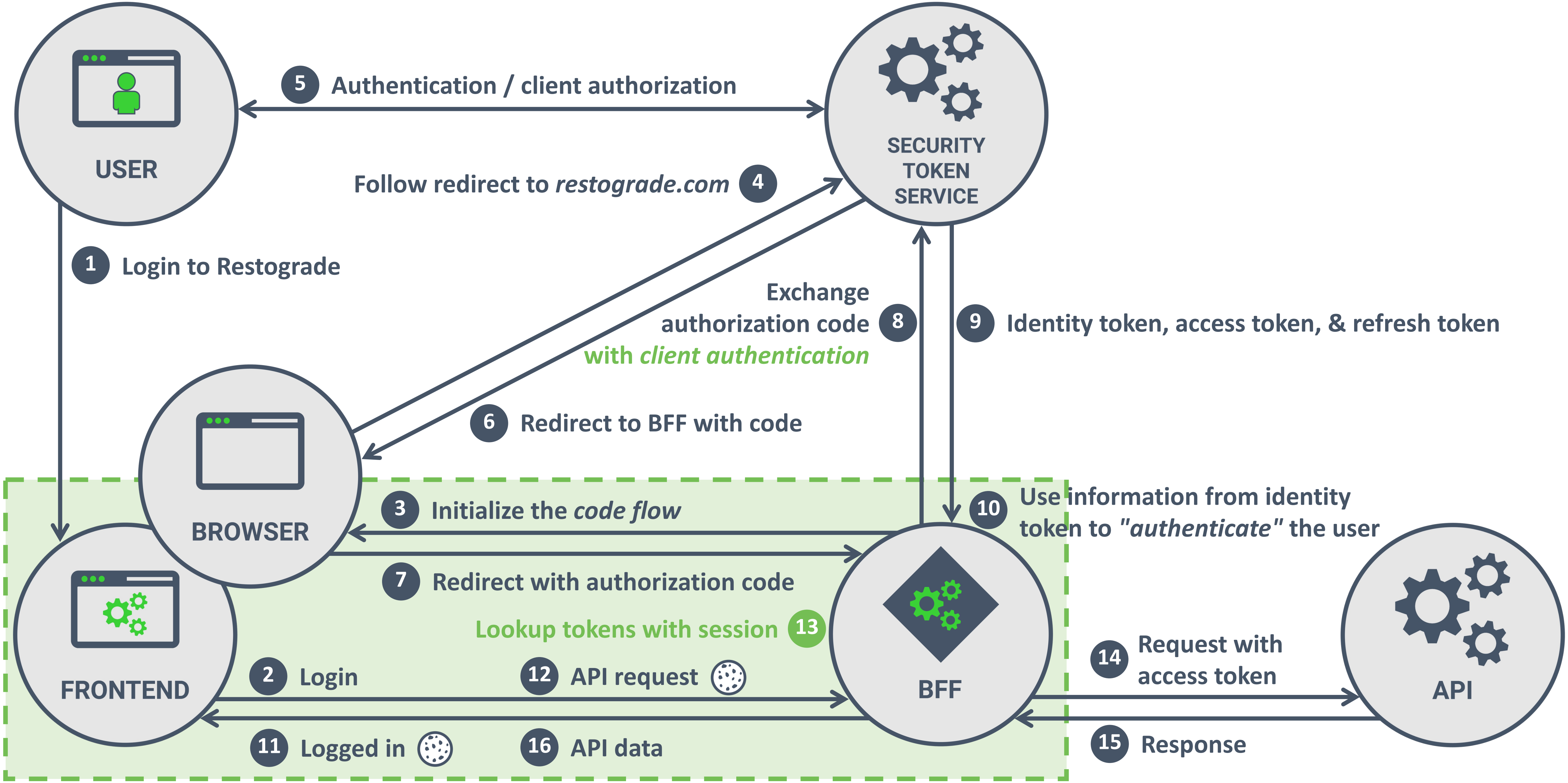
The attacker controlling the frontend can still impersonate the legitimate frontend and send requests to the BFF, which will be forward these requests to the APIs.

A BFF increases the security of OAuth 2.0

The application still consists of a frontend application interacting with APIs. The use of a BFF shifts the OAuth responsibilities from the browser to a server-side component.

A BFF-based architecture offers significant security improvements, while having a limited impact on the application.

SESSIONS BETWEEN THE FRONTEND AND THE BFF



USER
SECURITY TOKEN SERVICE
BROWSER
FRONTEND
BFF
API
1 Login to Restograde
2 Login
3 Initialize the *code flow*
4 Follow redirect to *restograde.com*
5 Authentication / client authorization
6 Redirect to BFF with code
7 Redirect with authorization code
8 Exchange authorization code with *client authentication*
9 Identity token, access token, & refresh token
10 Use information from identity token to *"authenticate"* the user
11 Logged in
12 API request
13 Lookup tokens with session
14 Request with access token
15 Response
16 API data

Is a BFF stateful or stateless?

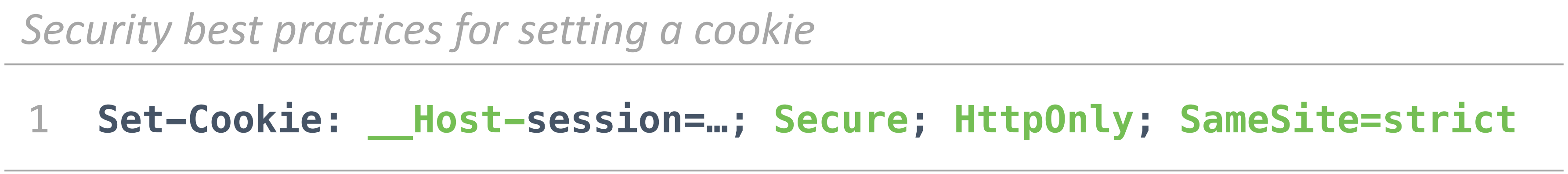
- BFF sessions can be implemented with or without server-side state
  - Server-side state keeps tokens on the server and issues a session ID in a cookie
  - Client-side state puts tokens into a session object and stores the object in a cookie

- Client-side sessions are often not recommended, due to lack of control
  - The session cookie has bearer token properties, so theft leads to abuse
  - Revoking existing state becomes difficult without server-side control over the session
  - In a BFF scenario, revocation is available through the OAuth 2.0 refresh tokens

- Client-side sessions in a BFF have strict security requirements
  - Confidentiality and integrity of this data is crucial to mitigate client-side attacks
  - Many server-side cookie frameworks support this out of the box

COOKIE SECURITY SETTINGS

- The BFF uses cookies to manage the session with the frontend
  - Browsers handle cookies automatically, so no need to write code in the frontend

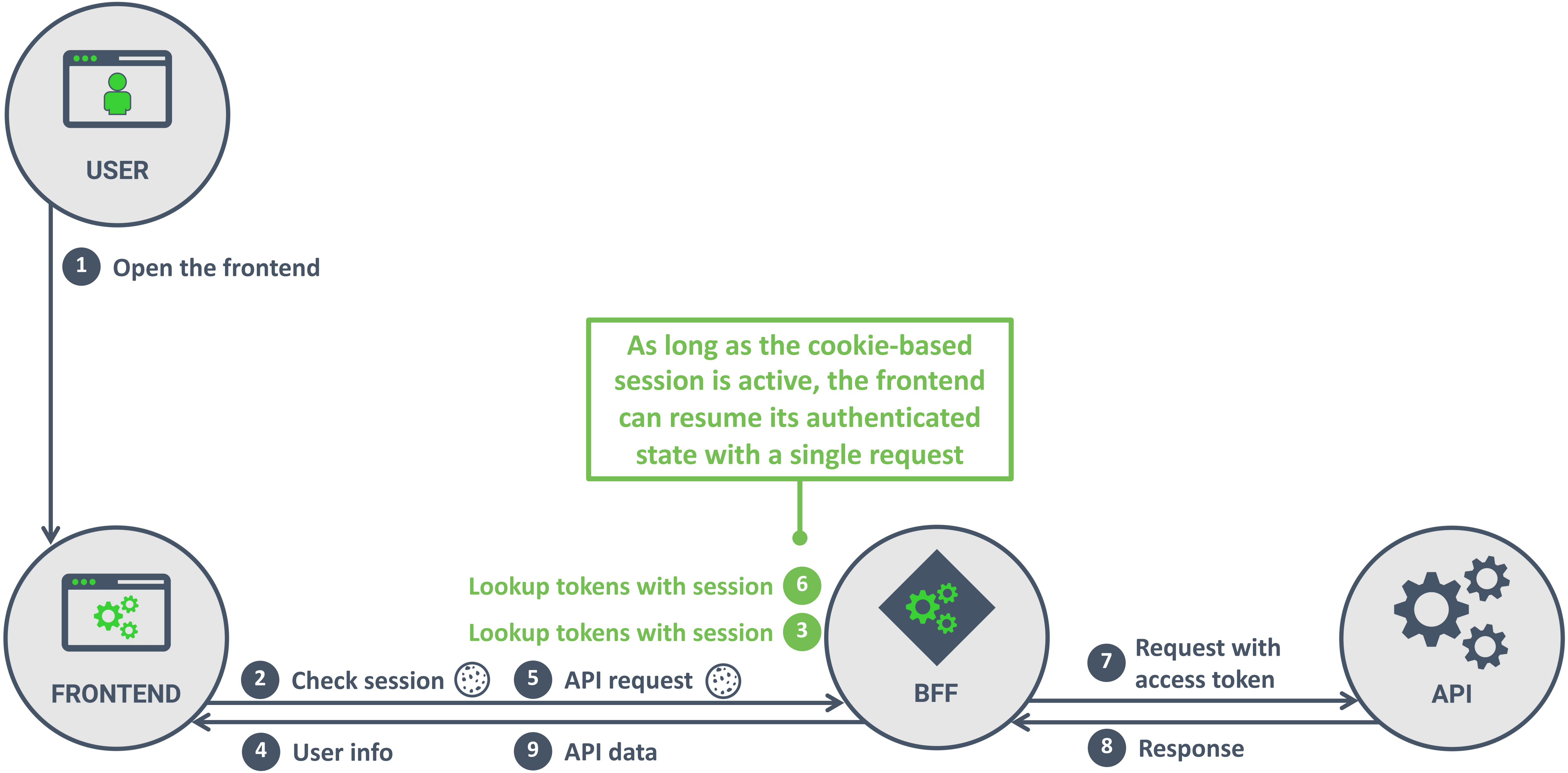


Security best practices for setting a cookie

```
Set-Cookie: __Host-session=…; Secure; HttpOnly; SameSite=strict
```

- Modern best practices for cookies require the following settings
  - Enable the ***Secure*** flag to restrict the cookie for HTTPS use only
  - Enable the ***HttpOnly*** flag to prevent JS-based access and memory-level attacks
  - Enable the ***SameSite=strict*** flag to prevent CSRF attacks
    - Only applies when the BFF is running on the same registered domain as the frontend
    - For cross-site frontend/BFF scenarios, remove this flag and configure CORS instead
  - Add the ***__Host-*** attribute to the name of the cookie to prevent subdomain-based attacks

Resuming a session with the BFF



USER

1 Open the frontend

FRONTEND

2 Check session

3 Lookup tokens with session

4 User info

5 API request

6 Lookup tokens with session

BFF

7 Request with access token

8 Response

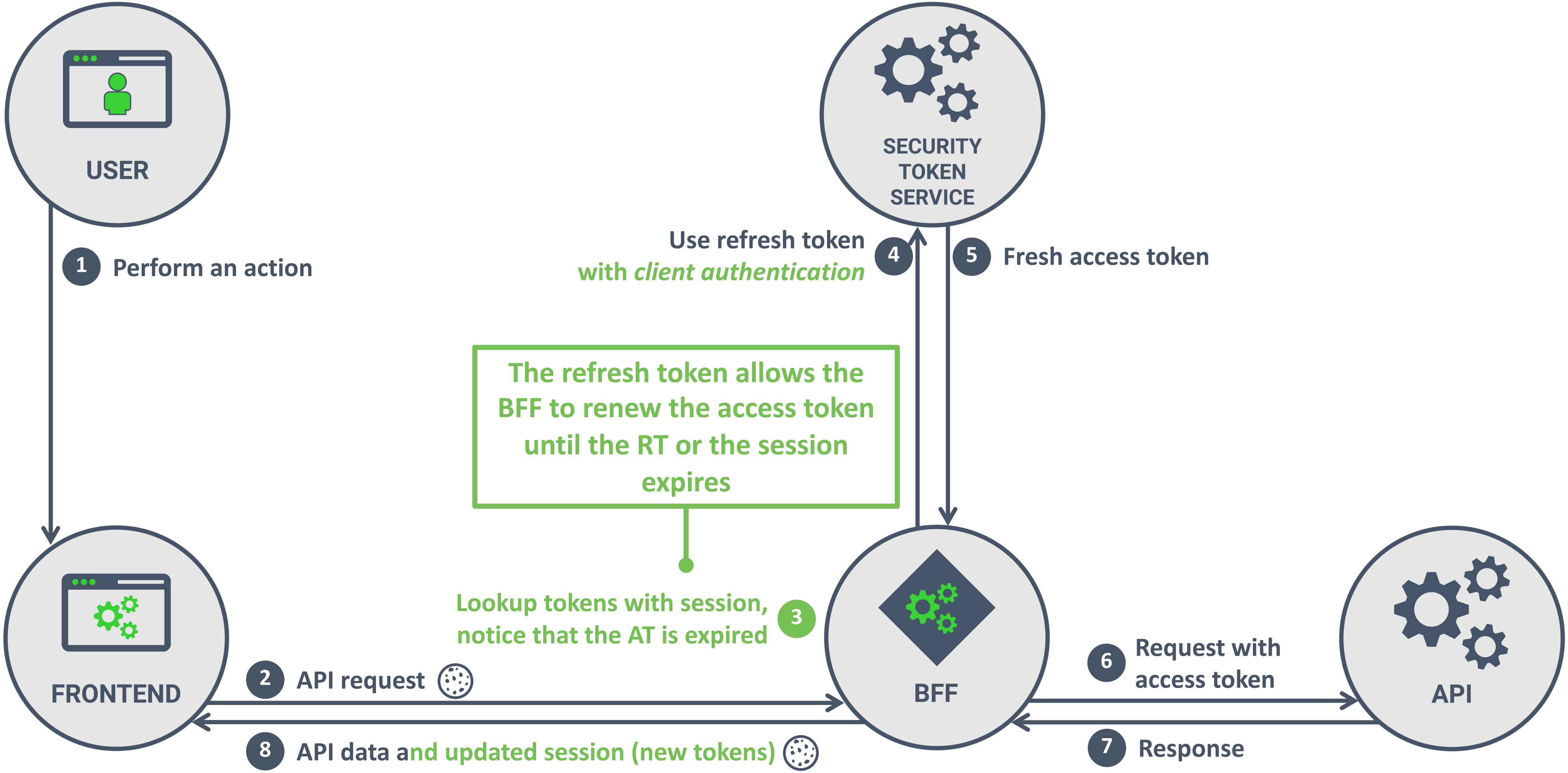
API

9 API data

As long as the cookie-based session is active, the frontend can resume its authenticated state with a single request

Implementing sessions in a BFF

Using refresh tokens with the BFF


USER
1 Perform an action
SECURITY TOKEN SERVICE
Use refresh token
with *client authentication*
4
5
Fresh access token
The refresh token allows the BFF to renew the access token until the RT or the session expires
Lookup tokens with session, notice that the AT is expired
3
FRONTEND
BFF
API
2 API request
8 API data and updated session (new tokens)
6 Request with access token
7 Response

Using refresh tokens with a BFF

Should the client-side cookie state be encrypted?

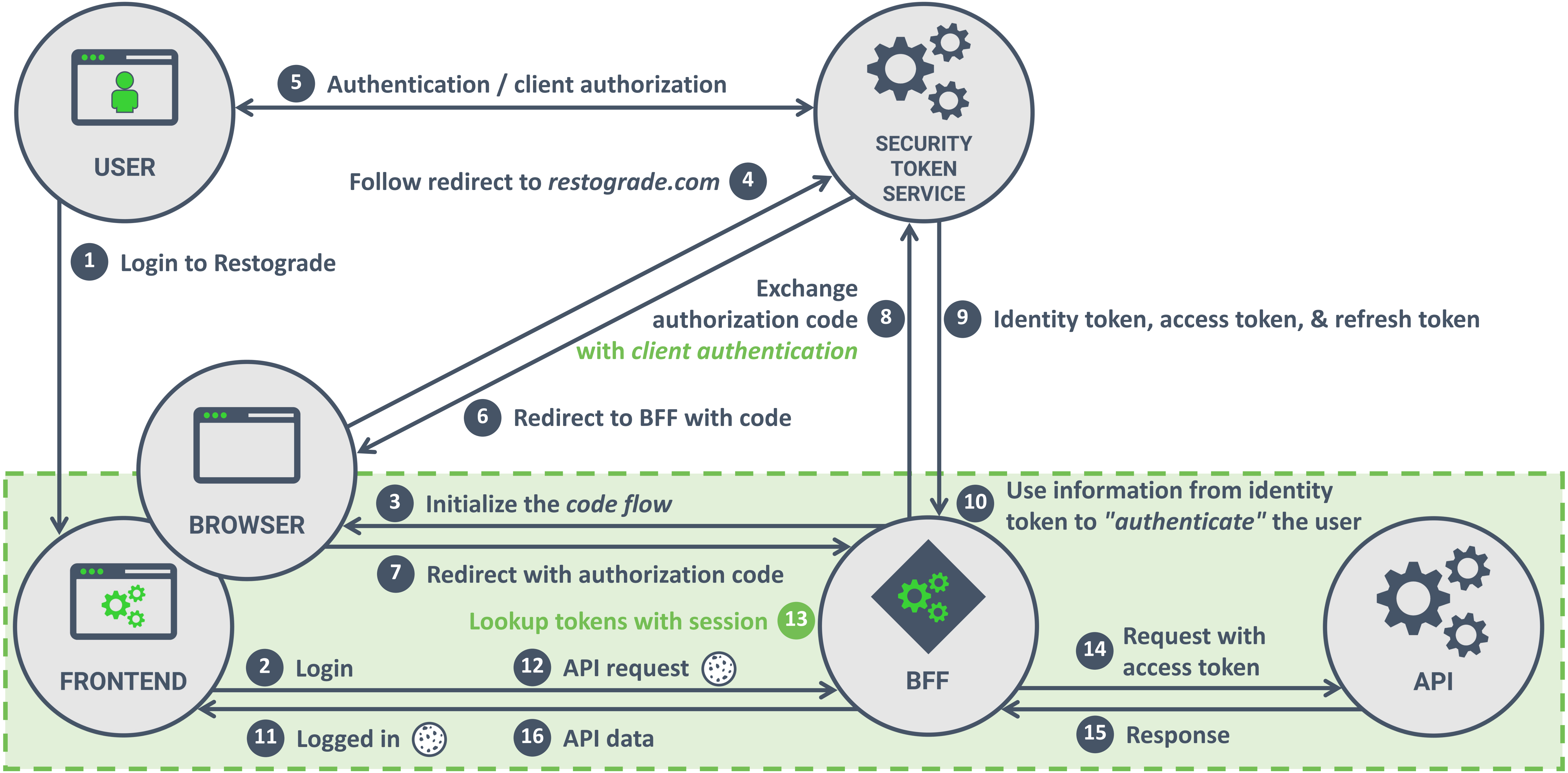
ENCRYPTING CLIENT-SIDE COOKIE STATE

- The client-side cookie state is created and read by the BFF
  - The BFF can choose to encrypt this state before sending it to the client
  - Encrypting the cookie state ensures full confidentiality on the client-side

- Encrypting session state is not mandatory to guarantee the security of the BFF
  - Cookies are configured to be inaccessible to the frontend application
  - An attacker executing code in the browser cannot get hold of the cookie state

- Advanced attack scenarios rely on external vectors to access browser state
  - E.g., Malware looking for Chrome profiles to steal cookies or access tokens
  - Encrypted cookie state can be used to counter such external attack vectors

PROXYING API REQUESTS


USER
SECURITY TOKEN SERVICE
BROWSER
FRONTEND
BFF
API
5 Authentication / client authorization
Follow redirect to *restograde.com* 4
1 Login to Restograde
Exchange authorization code with *client authentication* 8
9 Identity token, access token, & refresh token
6 Redirect to BFF with code
3 Initialize the *code flow*
10 Use information from identity token to "*authenticate*" the user
7 Redirect with authorization code
Lookup tokens with session 13
14 Request with access token
2 Login
12 API request
11 Logged in
16 API data
15 Response

The BFF as a proxy

The BFF becomes the OAuth 2.0 client application, in the name of the frontend

Each frontend that would have its own OAuth 2.0 client ID gets its own dedicated BFF

A BFF CONSISTS OF THREE CORE BUILDING BLOCKS

A BFF consists of three core building blocks, configured for a specific frontend/API interaction scenario.

A BFF consists of generic session management, OAuth 2.0 client responsibilities, and a proxy component.

Cookie-based applications need to mitigate CSRF attacks

CORS AS A CSRF DEFENSE

- The BFF can rely on CORS as a CSRF defense
  - It is crucial that every cross-origin request to the BFF API requires a CORS preflight
  - The BFF's policy does not approve this preflight, so the browser blocks the malicious call

- The simplest configuration for the BFF is to require a custom request header
  - When the attacker adds this header to a CSRF request, the browser enforces a preflight
  - A static header check is easy to implement and has no overhead

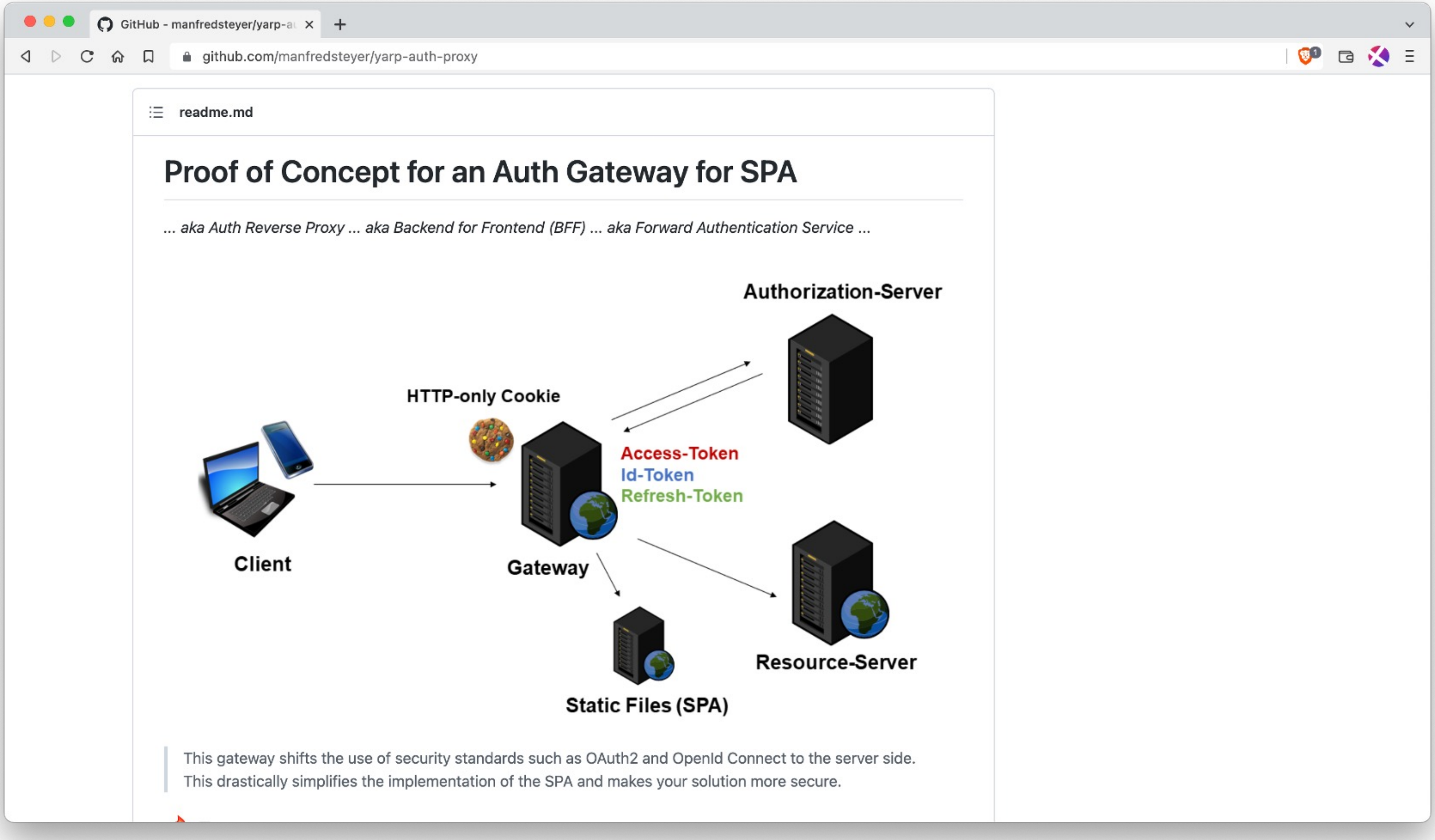
- CORS only applies on cross-origin requests
  - Legitimate same-origin interactions between frontend and BFF do not need preflights
  - Illegitimate cross-origin requests require a preflight and will be blocked by the browser

Adding CSRF defenses with CORS to the BFF

A BFF CAN HANDLE CSRF OUT OF THE BOX

A BFF can reject all cross-origin requests using custom middleware that validates the Origin *header.*

Alternatively, the BFF can require a custom request header on every request, allowing it to leverage a strict CORS policy as a CSRF defense.

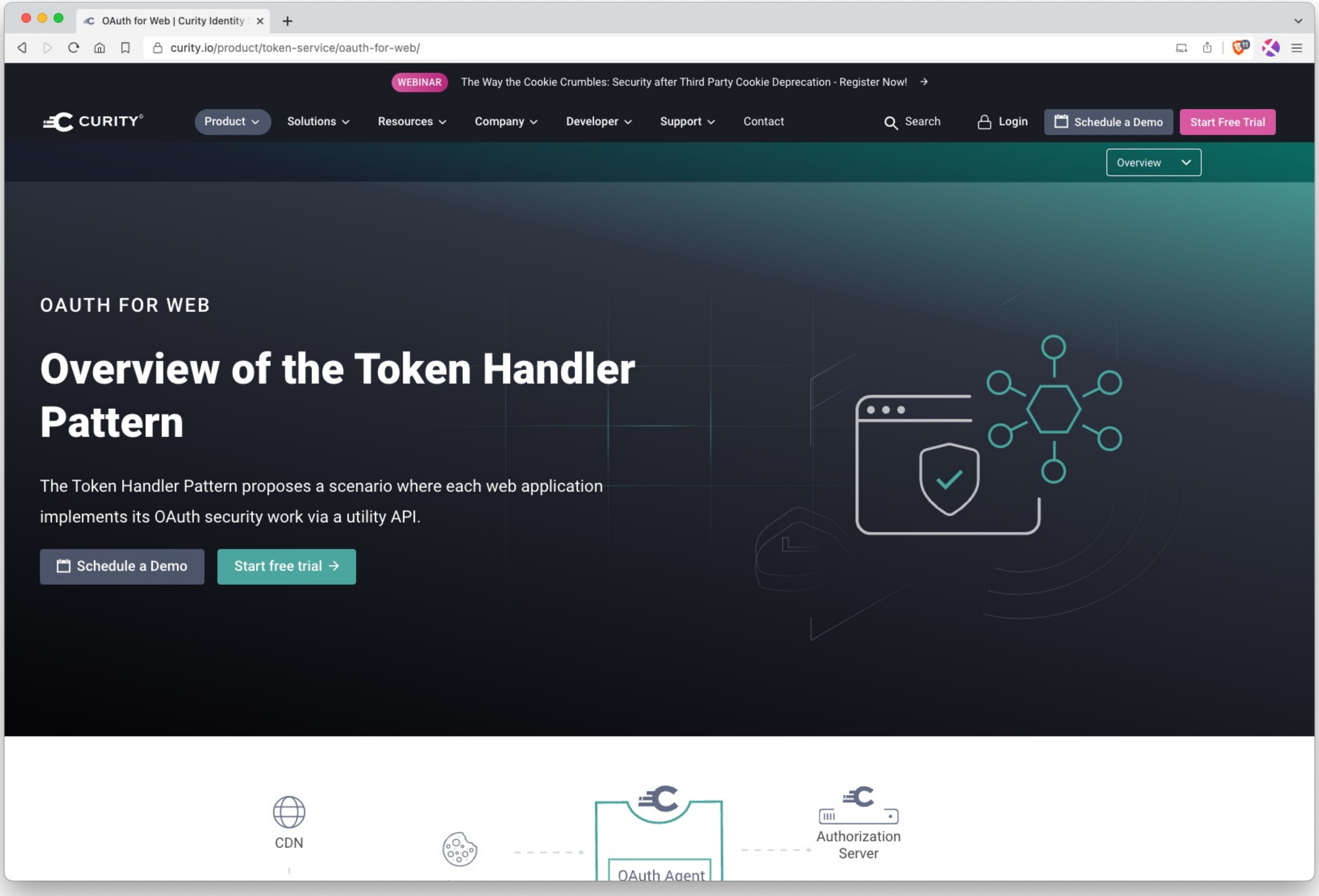

GitHub - manfredsteyer/yarp-a
github.com/manfredsteyer/yarp-auth-proxy
readme.md

Proof of Concept for an Auth Gateway for SPA

... aka Auth Reverse Proxy ... aka Backend for Frontend (BFF) ... aka Forward Authentication Service ...

> This gateway shifts the use of security standards such as OAuth2 and OpenId Connect to the server side. This drastically simplifies the implementation of the SPA and makes your solution more secure.

https://github.com/manfredsteyer/yarp-auth-proxy



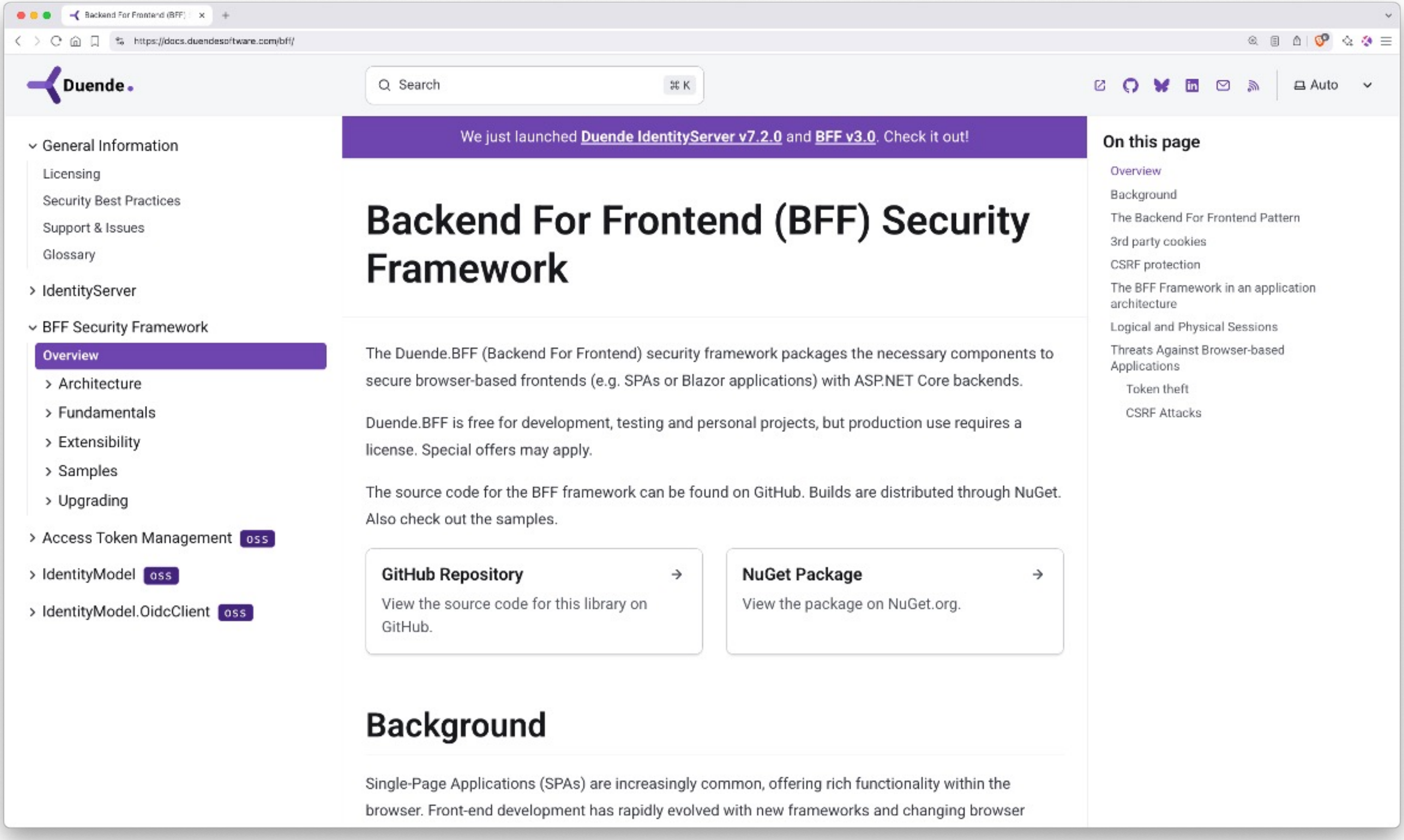
OAuth for Web | Curity Identity
curity.io/product/token-service/oauth-for-web/
WEBINAR
The Way the Cookie Crumbles: Security after Third Party Cookie Deprecation - Register Now!
CURITY
Product
Solutions
Resources
Company
Developer
Support
Contact
Search
Login
Schedule a Demo
Start Free Trial
Overview

OAUTH FOR WEB

Overview of the Token Handler Pattern

The Token Handler Pattern proposes a scenario where each web application implements its OAuth security work via a utility API.

https://curity.io/resources/learn/token-handler-overview/



We just launched **Duende IdentityServer v7.2.0** and **BFF v3.0**. Check it out!

Backend For Frontend (BFF) Security Framework

The Duende.BFF (Backend For Frontend) security framework packages the necessary components to secure browser-based frontends (e.g. SPAs or Blazor applications) with ASP.NET Core backends.

Duende.BFF is free for development, testing and personal projects, but production use requires a license. Special offers may apply.

The source code for the BFF framework can be found on GitHub. Builds are distributed through NuGet. Also check out the samples.

GitHub Repository →
View the source code for this library on GitHub.

NuGet Package →
View the package on NuGet.org.

Background

Single-Page Applications (SPAs) are increasingly common, offering rich functionality within the browser. Front-end development has rapidly evolved with new frameworks and changing browser

https://docs.duendesoftware.com/bff/

Sensitive applications should use a BFF

For sensitive applications, a BFF should be considered as the only secure option.

Scenarios that rely on a browser-based OAuth 2.0 client effectively adopt a "fingers crossed" security policy, hoping that the application never suffers from an attack able to run malicious JS code.

Key takeaways

1 Using OAuth 2.0 directly in the browser increases the attack surface

2 Use a BFF to simplify and optimize the security of your frontends

3 Follow secure coding guidelines to fix XSS in your applications

Thank you!

Need training or security guidance?
Reach out to discuss how I can help

https://pragmaticwebsecurity.com